Engenharia de Software para Ciência de Dados

Um guia de boas práticas com ênfase na construção de sistemas de Machine Learning em Python

Casa do Código | alura

MARCOS KALINOWSKI, TATIANA ESCOVEDO
HUGO VILLAMIZAR E HÉLIO LOPES

Sumário

- ISBN
- Sobre o livro
- Parte I – Introdução à Engenharia de Software e à Ciência de Dados
  - 1 Introdução à Engenharia de Software
  - 2 Introdução à Ciência de Dados
- Parte II – Abordagens e especificação de sistemas inteligentes
  - 3 Abordagens para a engenharia de sistemas inteligentes
  - 4 Especificação de sistemas de software inteligentes
- Parte III – Programação com boas práticas de projeto e construção
  - 5 Introdução à linguagem Python
  - 6 Orientação a Objetos em Python
  - 7 Boas práticas de projeto e construção de sistemas
- Parte IV – Tópicos de Ciência de Dados
  - 8 Análise exploratória e visualização de dados
  - 9 Pré-processamento de dados
  - 10 Algoritmos de Machine Learning para classificação e regressão
  - 11 Recursos avançados de Machine Learning
- Parte V – Arquitetura, Projeto e Controle da Qualidade
  - 12 Implantação de modelos de Machine Learning
  - 13 Arquitetura de sistemas de software inteligentes
  - 14 Projeto de sistemas de software inteligentes
  - 15 Controle da qualidade de sistemas de software inteligentes
  - 16 Gerência de configuração, DevOps e MLOps em sistemas inteligentes
  - 17 Referências bibliográficas

ISBN

Impresso: 978-85-5519-334-7

Digital: 978-85-5519-335-4

A arte da capa deste livro foi desenvolvida pela Noclaf.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Engenharia de software para ciência de dados : um guia de boas práticas com ênfase na construção de sistemas de machine learning / Marcos Kalinowski...[et al.]. -- 1. ed. -- São Paulo : Aovs Sistemas de Informática, 2023.

Outros autores: Tatiana Escovedo, Hugo Villamizar, Hélio Lopes.
Bibligorafia.
ISBN 978-85-5519-334-7

1. Ciência da computação 2. Dados - Estruturas (Ciência da computação) 3. Inteligência artificial 4. Engenharia de software 5. Processamento de dados I. Kalinowski, Marcos. II. Escovedo, Tatiana. III. Villamizar, Hugo. IV. Lopes, Hélio.

23-151608 CDD-005.1

Índices para catálogo sistemático:

1. Engenharia de software : Processamento de dados 005.1

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o livro

Os avanços recentes na área de Ciência de Dados — em especial em *Machine Learning* — e a disponibilidade de grandes quantidades de dados têm tornado viável e de relevância prática a incorporação de componentes de *Machine Learning* em sistemas de software. Esses componentes permitem às soluções de software aprender a partir dos dados e realizar predições inteligentes. Entretanto, sistemas que envolvem este tipo de componente muitas vezes são construídos sem considerar boas práticas da área de Engenharia de Software, levando muitos projetos de Ciência de Dados ao fracasso, com soluções que acabam não sendo implantadas na prática por não atenderem às necessidades dos clientes.

Com isso em vista, este livro foi cuidadosamente elaborado a partir de evidências científicas e experiências práticas dos autores em diversos projetos. O objetivo desta obra é estabelecer uma referência sobre Engenharia de Software para Ciência de Dados, compartilhando conhecimento (até então não consolidado) para capacitar profissionais interessados ou atuantes em Ciência de Dados na construção de sistemas baseados em *Machine Learning*, mostrando como construir esses sistemas *end-to-end*, adaptando e aplicando as melhores práticas da Engenharia de Software para esse contexto.

Você aprenderá a aplicar abordagens ágeis para a engenharia de sistemas inteligentes e a especificar e desenvolver sistemas baseados em *Machine Learning* na prática em Python, utilizando os principais algoritmos de classificação e regressão, seguindo princípios de projeto e boas práticas de codificação. Você aprenderá, ainda, a realizar o controle da qualidade de sistemas inteligentes, conhecerá alternativas para a arquitetura desses sistemas e diferentes formas de implantação de modelos. Por fim, conhecerá conceitos de gerência de configuração, DevOps e MLOps, comumente empregados nesse tipo de projeto.

O livro compila a vasta experiência dos autores envolvendo a entrega de dezenas de projetos de sistemas de software inteligentes na prática para diversas empresas, realizados por meio da iniciativa ExACTa PUC-Rio, incluindo soluções com pedidos de patente depositados e premiados (Reconhecimento à Inovação CENPES 2021, Inventor Petrobras 2022, entre outros). Os autores são pesquisadores consolidados nas áreas de Engenharia de Software e Ciência de Dados e pioneiros na interseção entre essas áreas, sendo protagonistas no estabelecimento da Engenharia de Software para Ciência de Dados no Brasil.

Em 2021, os autores criaram o primeiro curso de extensão em Engenharia de Software para Ciência de Dados do país, formando centenas de alunos em diversas turmas oferecidas pela PUC-Rio, tanto abertas quanto *in-company*. Atualmente, a disciplina faz parte do currículo das pós-graduações lato sensu em Ciência de Dados e Analytics e em Engenharia de Software e do elenco de disciplinas da pós-graduação stricto sensu (mestrado e doutorado) do Departamento de Informática da PUC-Rio. Além de auxiliar profissionais da área, este livro pode ser utilizado como referência bibliográfica para disciplinas com propósito similar em outras instituições.

Público-alvo e pré-requisitos

Este livro é indicado para profissionais das diversas áreas de conhecimento que gostariam de aprender a desenvolver aplicações de Ciência de Dados, em particular com ênfase em *Machine Learning*, seguindo boas práticas da Engenharia de Software. Isso inclui desenvolvedores e desenvolvedoras de software que queiram expandir seu conhecimento para Ciência de Dados; cientistas de

dados que queiram profissionalizar suas práticas de desenvolvimento de software; profissionais envolvidas(os) em iniciativas de transformação digital das suas empresas; e estudantes que desejem iniciar ou aprofundar seus estudos em Engenharia de Software para Ciência de Dados.

Para melhor aproveitar o conteúdo deste livro, é recomendado que o(a) leitor(a) tenha noções de conceitos matemáticos e de lógica de programação. Os conceitos teóricos apresentados serão complementados com exemplos práticos na linguagem Python.

Código-fonte

Todos os códigos e dados utilizados neste livro estão disponíveis no repositório do GitHub: https://github.com/profkalinowski/livroescd.

Sobre os autores

Marcos Kalinowski

Marcos Kalinowski é professor do Quadro Principal do Departamento de Informática da PUC-Rio, onde orienta pesquisas de mestrado e doutorado e coordena projetos de pesquisa e desenvolvimento junto a diversas empresas nas áreas de Engenharia de Software e Ciência de Dados por meio da iniciativa ExACTa PUC-Rio. É doutor e mestre em Engenharia de Sistemas e Computação na área de Engenharia de Software e bacharel em Ciência da Computação, todos pela UFRJ. Antes de se tornar professor, atuou por mais de 10 anos na indústria de software (como desenvolvedor, consultor e diretor). É bolsista de produtividade do CNPq e possui mais de 150 artigos publicados nos principais veículos da sua área de atuação. Atua como membro do International Software Engineering Research Network e como Senior Advisor da iniciativa nacional MPS.BR, que tem como objetivo melhorar a capacidade de desenvolvimento de software da indústria brasileira. Mais informações podem ser encontradas no LinkedIn

(https://www.linkedin.com/in/kalinowski/) e no Lattes (http://lattes.cnpq.br/1095304607841635).

Tatiana Escovedo

Tatiana Escovedo é professora do Departamento de Informática da PUC-Rio, onde coordena cursos de pós-graduação lato sensu, leciona em cursos de especialização, extensão e graduação e colabora com pesquisas nas áreas de Ciência de Dados e Engenharia de Software. Atua como gerente da área de Tecnologia, Gestão de Dados e Conhecimento da diretoria de Comercialização e Logística da Petrobras. É doutora em Engenharia Elétrica, na área de Métodos de Apoio à Decisão, mestre em Informática na área de Engenharia de Software e bacharel em Sistemas de Informação, todos pela PUC-Rio. É autora de diversos livros e artigos na sua área de atuação. Mais informações podem ser encontradas no LinkedIn (https://www.linkedin.com/in/tatiana-escovedo/) e no Lattes (http://lattes.cnpq.br/9742782503967999).

Hugo Villamizar

Hugo Villamizar é professor de cursos de extensão e especialização do Departamento de Informática da PUC-Rio. Atua como Analista de Pesquisa e Desenvolvimento na iniciativa ExACTa PUC-Rio, trabalhando com engenharia de software para sistemas inteligentes envolvendo *Machine Learning*, desde a especificação até sua implantação na nuvem. É doutorando e mestre em Informática pelo Departamento de Informática da PUC-Rio, orientado pelo Prof. Dr. Marcos Kalinowski, com dissertação premiada e diversos artigos na sua área de atuação. Graduado em Sistemas de Informação pela Universidade Nacional da Colômbia. Mais informações podem ser encontradas no LinkedIn (https://www.linkedin.com/in/hrguarinv/) e no Lattes (http://lattes.cnpq.br/8070469401750344).

Hélio Lopes

Hélio Lopes é professor do Quadro Principal do Departamento de Informática da PUC-Rio, onde orienta pesquisas de mestrado e doutorado e coordena projetos de pesquisa e desenvolvimento junto a diversas empresas nas áreas de Ciência de Dados e Engenharia de Software por meio da iniciativa ExACTa PUC-Rio. É doutor em Matemática, mestre em Informática e graduado em Engenharia da Computação, todos pela PUC-Rio. Tem mais de 30 anos de experiência coordenando projetos de pesquisa e desenvolvimento junto a diversas empresas, tendo registrado patentes e acumulado premiações nesse contexto. É bolsista de produtividade do CNPq e possui mais de 150 artigos publicados nos principais veículos da sua área de atuação. Mais informações podem ser encontradas no LinkedIn (https://www.linkedin.com/in/helio-cortes-vieira-lopes) e no Lattes (http://lattes.cnpq.br/9199970180870105).

Agradecimentos

Muitas foram as pessoas envolvidas de alguma forma no aprendizado para a elaboração deste livro e, por isso, não faremos agradecimentos nominais individuais, para evitar o risco de sermos injustos se nos esquecermos de alguém. Mas gostaríamos de deixar alguns agradecimentos registrados:

- Aos colaboradores da iniciativa ExACTa PUC-Rio e das empresas parceiras. O aprendizado prático adquirido no contexto da entrega de dezenas de soluções reais foi fundamental para a escrita deste livro. Um pouco deste livro foi escrito por cada um de vocês;

- Aos colaboradores de pesquisa do Software Science Lab e do Data Science Lab (DASLAB), incluindo nossos alunos de mestrado e doutorado e, também, colaboradores nacionais e internacionais. As pesquisas desenvolvidas neste contexto nos permitiram o aprofundamento necessário para a escrita do livro. Diversos dos conhecimentos trazidos foram obtidos nessas pesquisas e muitas delas se encontram nas nossas referências;

- Aos profissionais das primeiras turmas (abertas e *in-company*) dos cursos de Engenharia de Software para Ciência de Dados. O retorno positivo recebido foi motivador para a consolidação do livro e nos deu a certeza da importância e da utilidade prática da compilação deste material;

- Aos demais professores do Departamento de Informática da PUC-Rio, pelo apoio incondicional e pelo estabelecimento de um ambiente propício à inovação e à pesquisa de qualidade;

- Aos professores que contribuíram com a nossa formação acadêmica e profissional. Vocês serviram como referência e nos

ajudaram a criar asas para buscar novos conhecimentos e alçar os nossos próprios voos com autonomia intelectual.

Por último, mas não menos importante, gostaríamos de registrar nossos agradecimentos a nossas famílias e amigos pela compreensão em relação ao tempo dedicado a este projeto.

Parte I – Introdução à Engenharia de Software e à Ciência de Dados

Capítulo 1
Introdução à Engenharia de Software

Para aqueles que vêm de uma área de conhecimento diferente da Computação, a primeira coisa que devem estar se questionando é: o que é essa tal de Engenharia de Software? De forma resumida, Engenharia de Software é a aplicação de princípios usados no campo da Engenharia, que geralmente lida com sistemas físicos, para a especificação, projeto, desenvolvimento, teste, implantação e gerenciamento de sistemas de software.

Considerando a construção de sistemas de software, programação é uma parte importante do processo, mas não é tudo! Precisamos também investir esforços em entender o que programar (requisitos), como programar (arquitetura e projeto), se o que foi programado está certo (revisões e teste de software) e como implantar a solução de forma eficiente em operação (*DevOps*).

1.1 Precisamos de Engenharia de Software?

Há diversos fatores envolvidos na produção de software do "mundo real", como **custo**, **prazo** e **qualidade**. Em função do tamanho do software, esses fatores podem ser difíceis de garantir! É muito mais fácil, por exemplo, estimar o esforço para o desenvolvimento de uma pequena agenda pessoal do que para um sistema de gestão integrada envolvendo dados de várias áreas de negócio.

De fato, há diversas evidências científicas que apontam benefícios da aplicação de boas práticas da Engenharia de Software no custo, prazo e qualidade de projetos de software. Em uma investigação realizada ao longo de oito anos junto a centenas de empresas desenvolvedoras de software (TRAVASSOS; KALINOWSKI, 2014), observamos benefícios de aumento da produtividade e de redução de retrabalho decorrentes de investimentos em princípios de engenharia para desenvolver um software correto por construção. Ademais, os esforços de manutenção tendem a ser significativamente reduzidos quando um software é construído com os devidos cuidados de arquitetura e projeto da solução.

Estima-se que, em projetos de software, o esforço de retrabalho esteja entre 30% e 40% do esforço total do projeto. Isso seria inimaginável para projetos em áreas como a Engenharia Civil. Assim, a aplicação de boas práticas da Engenharia de Software costuma dar resultados imediatos. Por exemplo, recentemente aumentamos a produtividade de uma das nossas empresas parceiras ao reduzir o retrabalho para um terço do que era antes da nossa intervenção. Basicamente o que fizemos foi trazer boas práticas para a especificação dos sistemas e o seu controle da qualidade.

1.2 Precisamos de Engenharia de Software para Ciência de Dados?

Sabemos que, hoje, soluções baseadas em software permeiam as nossas vidas em quase todos os aspectos. Além disso, os avanços contemporâneos em *Machine Learning* (ML) e a disponibilidade de grandes quantidades de dados têm tornado viável e de relevância prática incorporar componentes de ML em sistemas de software. Esses componentes permitem às soluções de software aprender a partir dos dados e realizar classificações, predições, agrupamentos e associações inteligentes. Daqui em diante, chamaremos esses sistemas de **Sistemas de Software Inteligentes**.

Anos atrás, havia um questionamento se fazer software não era mais arte do que engenharia, considerando que a criatividade da pessoa programadora é uma parte importante do processo. Esse mito foi desconstruído para sistemas convencionais com base em evidências diversas, reforçando a importância e os benefícios da Engenharia de Software. Hoje, é comum escutarmos o mesmo argumento para sistemas de software inteligentes, em particular para a construção dos componentes de ML, que muitas vezes envolve modelos construídos por profissionais de outras áreas que não a Computação em si, como biólogos, economistas, estatísticos, entre outros.

Ao comparar software com arte, precisamos pensar nas consequências de um software conter defeitos que resultem em falhas quando submetido à operação. Em geral, soluções são embutidas em sistemas de software para permitir sua sistematização e aplicação em larga escala. Dessa forma, softwares que apresentam falhas em operação podem literalmente sistematizar o caos. Embora muitas vezes isso não seja claro para a sociedade como um todo, as consequências de falhas podem ser ainda mais drásticas no caso de sistemas de software inteligentes.

Como primeiro exemplo, podemos recordar o desastre de Fukushima de 2011, o segundo pior desastre nuclear da história, com um

terremoto de magnitude 9 que devastou a usina, teve 154 mil pessoas evacuadas e mais de 18 mil mortes. O que subsidiou a decisão de se construir uma usina nuclear que suportasse um terremoto de magnitude 8,6 foi um modelo de *Machine Learning* de regressão linear, construído em cima de dados de treinamento, mas que se adaptou em excesso a esses dados (o que conhecemos como *overfitting*) e fez a predição de um terremoto de magnitude 9 a cada 13 mil anos. Se o modelo fosse construído sem o *overfitting*, seria esperado um terremoto de magnitude 9 a cada 300 anos, e certamente a usina teria sido construída de forma a se preparar adequadamente.

Outro exemplo foi um caso ocorrido com um cientista de dados no Rio de Janeiro, que foi preso por engano após um sistema de reconhecimento facial tê-lo identificado equivocadamente como um miliciano. Ele ficou preso por 22 dias até que o problema fosse detectado. Caso similar ocorreu também em Detroit, exemplificando sistemas que não apresentam a qualidade que deveriam ter. Nesses casos há, ainda, questões éticas no uso desse tipo de sistema, já que sistemas de reconhecimento facial tendem a ter mais facilidade na classificação para pessoas de pele mais clara. Aspectos éticos devem ser sempre considerados na construção de sistemas de software inteligentes, evitando que se cometam injustiças.

Finalmente, não é difícil encontrar notícias a respeito de acidentes envolvendo carros autônomos, em alguns casos provocando mortes de pessoas. De fato, imaginem as consequências de termos inúmeros carros autônomos com falhas em seus modelos de tomada de decisão rodando nas ruas. É um exemplo da sistematização do caos que mencionamos anteriormente.

Entretanto, temos também exemplos de softwares inteligentes bem-sucedidos, como o Smart Tocha, desenvolvido na iniciativa ExACTa PUC-Rio em cocriação com a Petrobras e que teve seu pedido de patente depositado recentemente, além de ter recebido o prêmio Inventor Petrobras 2022. O Smart Tocha aplica técnicas de análise de imagens e aprendizado de máquina em imagens da queima na

tocha das refinarias para atuar automaticamente no sistema de controle para regular a injeção de vapor. A implantação desta solução (hoje presente em diversas refinarias), elaborada seguindo boas práticas da Engenharia de Software apresentadas neste livro, está gerando, por refinaria, uma economia de energia constante equivalente ao consumo de uma cidade de 20 mil habitantes.

Além dos riscos já exemplificados, há impactos econômicos também na manutenção desses sistemas. Sculley *et al.* (2014), no artigo *Machine Learning: The High-Interest Credit Card of Technical Debt*, chamam a atenção para a facilidade de gerar custos extremamente elevados de manutenção nesse tipo de sistema. Na prática, apenas uma pequena parte do código de um sistema de software inteligente é composta pelo código específico de *Machine Learning*, mas a infraestrutura de software requerida para o sistema como um todo tende a ser vasta e complexa (SCULLEY *et al.*, 2015) e deve ser pensada aplicando as melhores práticas de Engenharia de Software para viabilizar sua manutenção. Cabe ressaltar ainda que, embora normalmente corresponda à menor parte do sistema, o código de *Machine Learning* tende a ser o mais frequentemente modificado na prática (TANG *et al.*, 2021). De fato, são muitos os problemas e dificuldades reportados pelas empresas para a construção de sistemas de software inteligentes (KALINOWSKI *et al.*, 2023).

Enquanto sistemas de software inteligentes que envolvem aprendizado de máquina são de fato sistemas de software, suas características os tornam difíceis de testar e muitas vezes o que sabemos sobre como projetar sistemas não se aplica diretamente à engenharia de sistemas de software inteligentes (OZKAYA, 2020). Sem dúvida, a Engenharia de Software pode apoiar a especificação, a arquitetura, o projeto, a construção e o controle da qualidade de sistemas de software inteligentes, melhorando a qualidade desses sistemas e evitando consequências potencialmente desastrosas para nossa sociedade, como as exemplificadas. Entretanto, as práticas da Engenharia de Software tradicional claramente necessitam de adaptações para que possam ser aplicadas neste contexto, o que

fica ainda mais evidente com a recente inclusão da Engenharia de Software para sistemas inteligentes envolvendo inteligência artificial e aprendizado de máquina na agenda nacional prioritária americana para pesquisa em Engenharia de Software (CARLETON *et al.*, 2022).

Considerando esta necessidade, este livro é o primeiro a buscar abordar *end-to-end* como a Engenharia de Software pode ser aplicada para sistemas inteligentes, desde a especificação dos sistemas até a sua implantação. O conteúdo considera pesquisas de vanguarda na interseção entre Engenharia de Software e Ciência de Dados e experiências práticas vivenciadas na iniciativa ExACTa PUC-Rio, que produziu e entregou dezenas de sistemas inteligentes para parceiros da indústria de diferentes setores de negócio. Estamos convictos de que o conteúdo aqui apresentado poderá ajudar a melhorar a prática profissional nesta área tão importante, auxiliando as empresas de software a melhorar a qualidade dos sistemas inteligentes que constroem para a sociedade.

1.3 Processo de Engenharia de Software

Uma das metas da Engenharia de Software é definir uma abordagem ou um processo, isto é, definir a maneira como o software será construído, incluindo as boas práticas a serem seguidas durante a construção. Assim, o processo de Engenharia de Software tem como objetivo garantir a produção de software de alta qualidade em acordo com as necessidades dos seus usuários finais (escopo adequado), com cronograma e custo previsíveis.

Normalmente esse processo é composto por um conjunto de atividades que seguem um modelo de ciclo de vida, bem definidas, com dependências entre si e ordem de execução, com responsáveis e artefatos de entrada e saída. É desejável ainda que as atividades tenham uma descrição sistemática das tarefas a serem realizadas.

Os modelos de ciclo de vida mais amplamente conhecidos são:

- Modelo Cascata;
- Modelo Incremental;
- Modelo Evolutivo;
- Ciclo de vida associado ao Scrum.

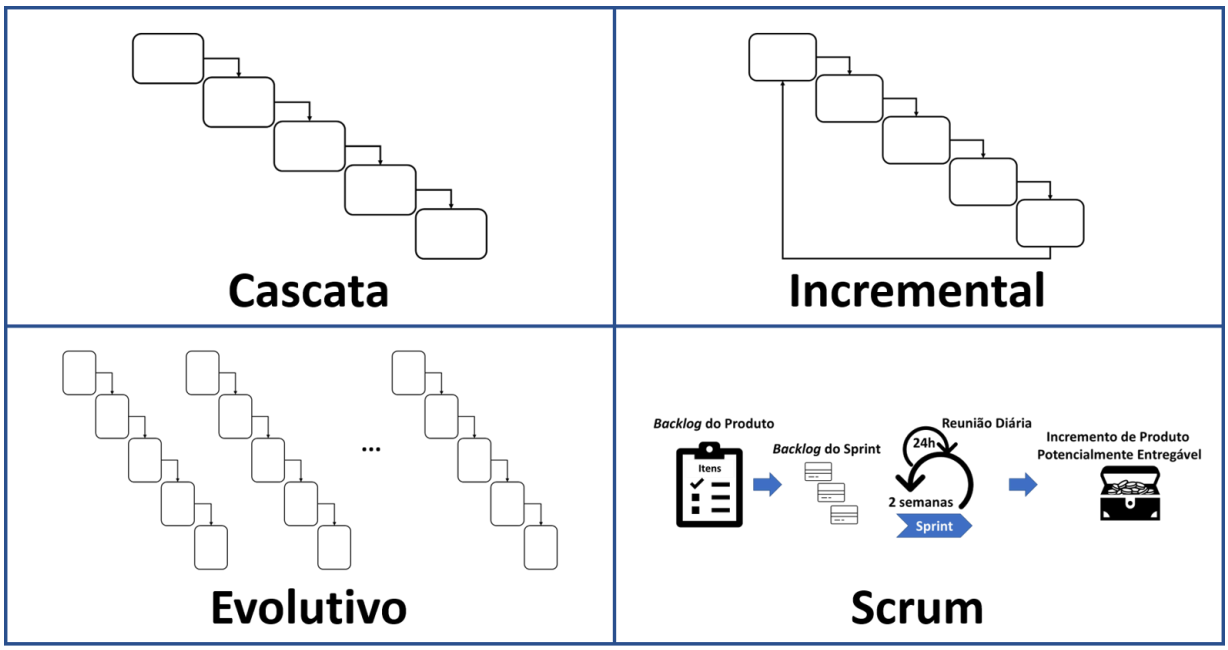


Figura 1.1: Modelos de ciclo de vida mais conhecidos.

O **modelo cascata** assume a execução sequencial das atividades e, em geral, se mostra adequado apenas para projetos pequenos — por exemplo, que envolvam menos de 200 horas. Isso porque ele presume que é possível realizar toda a especificação de uma só vez, validá-la e então seguir para a atividade seguinte. Na prática, essa natureza sequencial raramente consegue ser seguida em projetos de médio ou grande porte. O modelo *incremental* e o *evolutivo* apresentam alternativas.

No **modelo incremental**, os requisitos são segmentados em uma série de *incrementos*. A cada incremento, é produzida uma versão operacional do software, e o processo se repete até que um produto completo seja produzido. A segmentação de requisitos é usualmente

planejada *a priori*. Dessa forma, um menor custo e menos tempo são necessários para se entregar a primeira versão, e os riscos associados ao desenvolvimento são menores, devido ao tamanho reduzido de cada incremento. O número de solicitações de mudança nos requisitos também pode diminuir, devido ao curto tempo de desenvolvimento dos incrementos. Esse é o modelo de ciclo de vida utilizado nas variantes do *processo unificado*.

Processo unificado é um nome genérico para uma família de modelos de processos orientados a plano iterativos (as iterações são planejadas *a priori*) e incrementais. O mais conhecido entre esses modelos de processo é o *Rational Unified Process* (RUP), que é um refinamento do processo unificado criado pela Rational Software (atualmente da IBM).

A figura a seguir ilustra a dinâmica do modelo incremental. Nesta figura, um pintor faz entregas parciais de uma pintura completa. Note que cada parte é "entregue" de forma completamente finalizada.

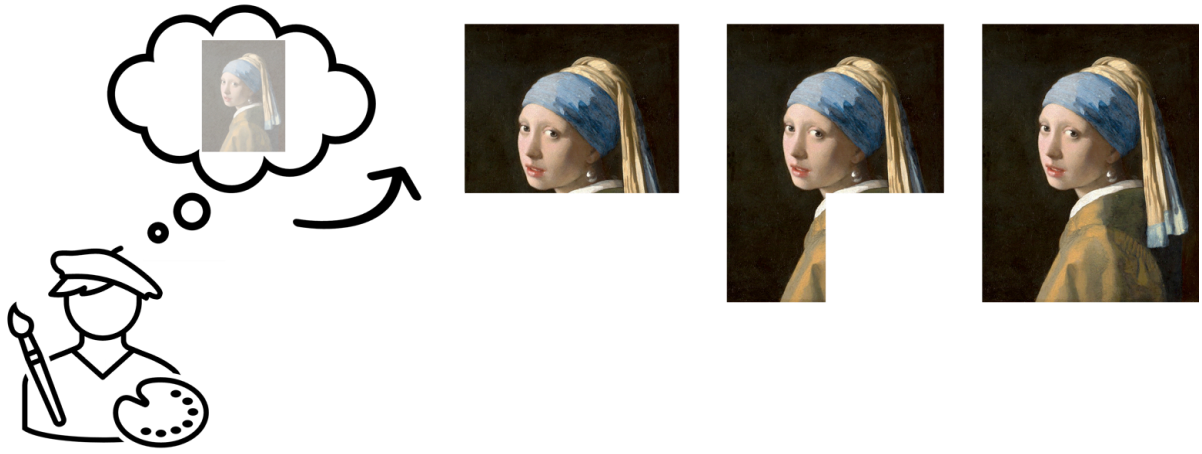


Figura 1.2: Exemplo de modelo incremental.

No **modelo evolutivo**, são desenvolvidas versões parciais que atendem aos requisitos conhecidos inicialmente. A primeira versão é então usada para refinar os requisitos para uma segunda versão, e assim por diante. A partir do conhecimento sobre os requisitos

obtido com o uso, continua-se o desenvolvimento, evoluindo o produto. Esse ciclo de vida tende a se mostrar adequado quando os requisitos não podem ser completamente especificados de início, e o uso do sistema pode aumentar o conhecimento sobre o produto e melhorar os requisitos. Mas é preciso ter cuidado na sua aplicação. Além de poder gerar muito retrabalho, os usuários podem não entender a natureza da abordagem e se decepcionar quando os resultados não são satisfatórios.

A figura a seguir ilustra a dinâmica do modelo evolutivo. Nesta figura, um pintor faz evoluções recorrentes em uma pintura completa. Note que cada entrega é uma evolução da versão anterior.

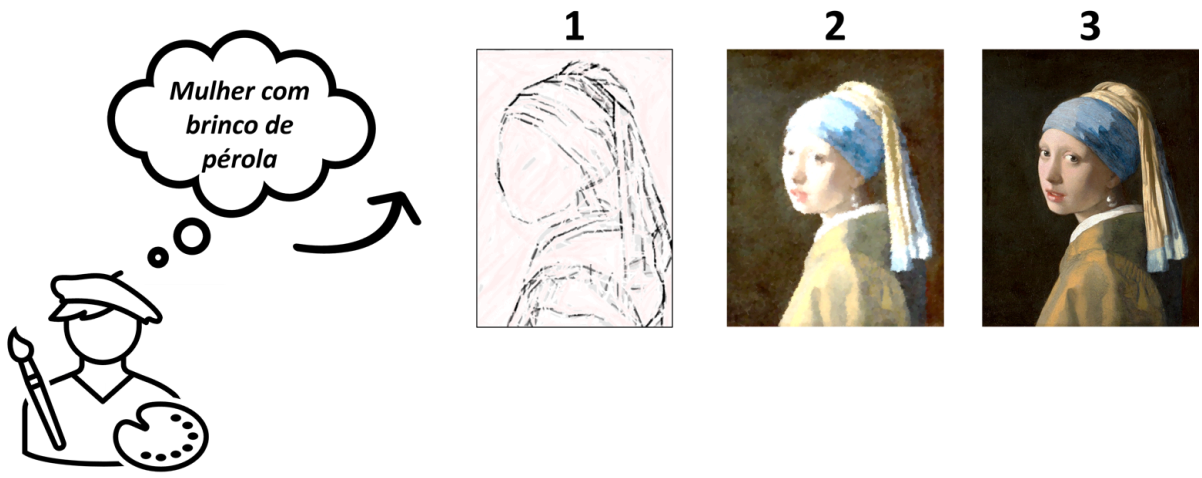


Figura 1.3: Exemplo de modelo evolutivo.

O **ciclo de vida associado ao Scrum** é o mais amplamente utilizado atualmente. Por essa razão, tendo em vista a formação objetiva e focada no mercado, ele será o mais detalhado neste livro. De fato, teremos uma seção inteira sobre a *gestão ágil de projetos de software* nos próximos capítulos, mas, por enquanto, basta compreender a natureza desse ciclo de vida.

O **Scrum** é um *framework* de gestão ágil iterativo e incremental / evolutivo bastante aplicado no contexto de desenvolvimento de software. A dinâmica do Scrum está ilustrada na figura a seguir.

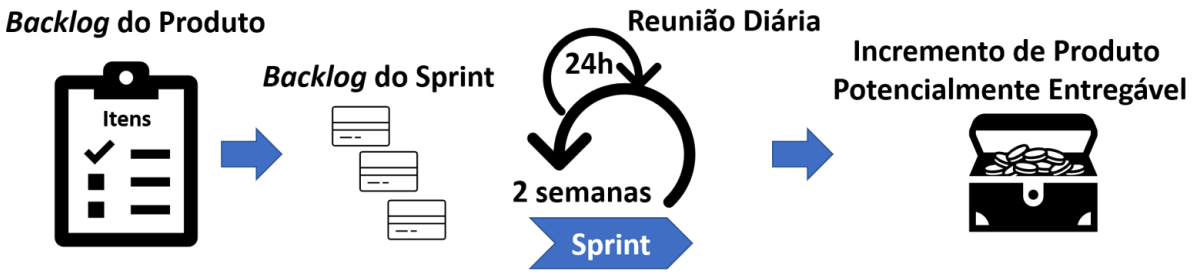


Figura 1.4: Ciclo de vida associado ao Scrum.

É possível ver que o ciclo de vida do Scrum utiliza tempo fixo (***sprints***) em vez de escopo fixo para determinar seus incrementos. Cada *sprint* corresponde a uma iteração que entrega incrementos e/ou evoluções de um software, conforme prioridade planejada para agregar valor ao negócio. Um ***backlog*** do produto é um conjunto de requisitos, priorizado pelo *Product Owner*, responsável por conhecer as necessidades do cliente.

Entregas de um conjunto fixo de itens do *backlog* ocorrem em *sprints*. Os itens do *backlog* que serão tratados em cada *sprint* (***backlog* do *sprint***) são definidos em uma sessão de planejamento do *sprint*. Durante o *sprint*, ocorrem breves **reuniões diárias**, em que cada participante fala sobre o progresso conseguido, o trabalho a ser realizado e/ou o que o(a) impede de seguir avançando. Ao final de um *sprint*, ocorre a **revisão** e a **retrospectiva**, nas quais todos os membros da equipe revisam a entrega e refletem sobre o *sprint* passado. O Scrum é facilitado por um *Scrum Master*, que tem como função primária remover qualquer impedimento à habilidade de uma equipe de entregar o objetivo do *sprint*.

Agora que já compreendemos as dinâmicas típicas de processos e abordagens de desenvolvimento de software, veremos que tipos de práticas mais específicas a Engenharia de Software contempla.

1.4 Práticas de Engenharia de Software

Existem diversos modelos de referência, normas e corpos de conhecimento que organizam práticas de Engenharia de Software.

Entre os modelos de referência, destacamos o modelo de referência brasileiro, que é o **Modelo MPS para Software (MPS-SW)**, do programa para a Melhoria do Processo de Software Brasileiro (MPS.BR), e o ***Capability Maturity Model Integration* (CMMI)**, da Information Systems Audit and Control Association (ISACA), principal modelo de referência internacional.

Entre as normas, a **ISO 12207** – *Systems and Software Engineering - Software Life Cycle Processes* – representa a norma ISO mais significativa para a Engenharia de Software. Além disso, existe um corpo de conhecimento organizado pela IEEE Computer Society, o ***Software Engineering Body of Knowledge (SWEBOK)***.

Como existe uma certa correspondência e compatibilidade entre as práticas propostas por esses modelos, normas e corpos de conhecimento, ilustraremos as práticas de Engenharia de Software com base no MPS-SW. É importante ressaltar que existe evidência científica de que os resultados da adoção de boas práticas contidas nesses tipos de modelo de referência tendem a ser positivos na prática (KALINOWSKI *et al.*, 2015).

Essencialmente, o MPS-SW contempla boas práticas da Engenharia de Software, organizadas em processos. A figura a seguir, extraída do Guia Geral de MPS para Software de 2021 (SOFTEX, 2021), apresenta os **processos de projeto** e os **processos organizacionais** desse modelo. Esses processos fornecem uma boa visão geral de tópicos relevantes para o mercado estudados no contexto da Engenharia de Software.

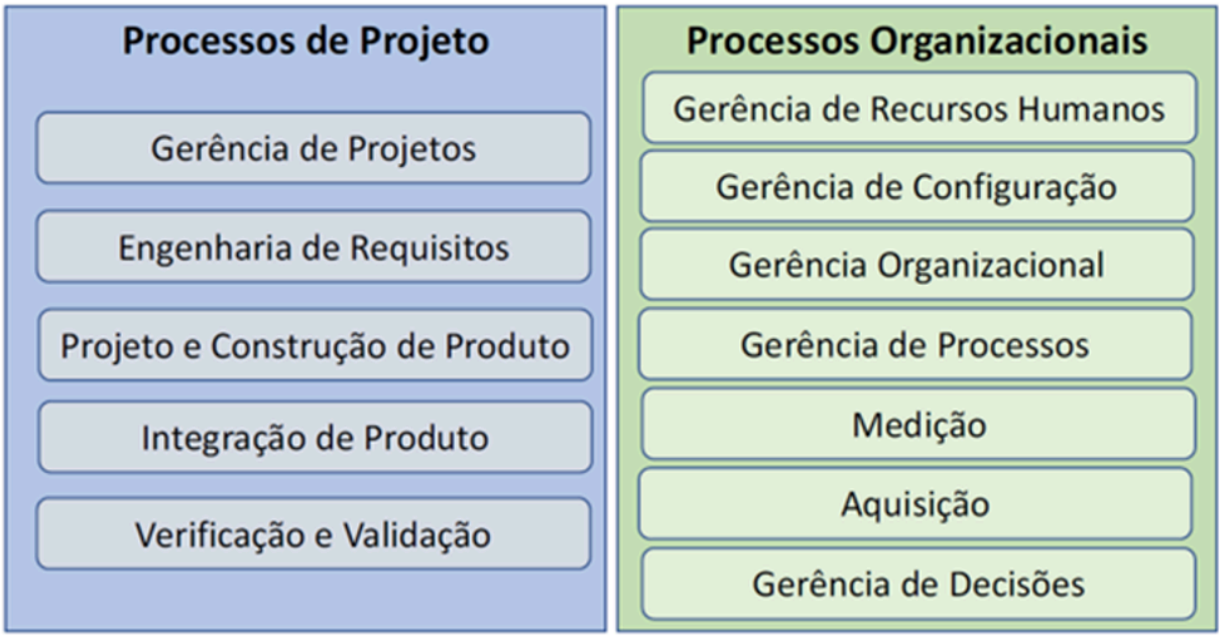


Figura 1.5: Processos de projeto e processos organizacionais do MPS-SW. Fonte: Softex, 2021.

É possível observar que os *processos de projeto* contemplam processos mais voltados para a gestão e a engenharia do software, enquanto os *processos organizacionais* podem ser vistos como processos de apoio para o bom funcionamento dos projetos.

Em relação aos processos de projeto, eles organizam práticas de **Gerência de Projetos** (incluindo práticas de planejamento e estimativa e para acompanhamento dos projetos), **Engenharia de Requisitos** (incluindo práticas para a especificação de software e a gestão de mudanças), **Projeto e Construção de Produto** (incluindo práticas relacionadas com a modelagem, codificação e testes unitários de componentes), **Integração do Produto** (incluindo práticas para determinar a estratégia de integração de componentes e os testes de integração) e **Verificação e Validação** (incluindo práticas de revisão e teste de sistemas e de aceitação). Ao longo do livro, daremos uma visão geral dessas práticas com ênfase na adaptação para a sua aplicação moderna na construção de sistemas de software inteligentes.

Em relação aos processos organizacionais, enfatizaremos aspectos mais práticos relacionados com a **Gerência de Configuração** (incluindo práticas de controle de versões e de mudanças), por ser um processo fundamental para o bom andamento de projetos de software. Informações para os demais processos poderão ser encontradas no modelo, que tem seus guias gratuitamente disponíveis on-line.

Um problema comum é confundir os modelos de referência com métodos. Os modelos não limitam os métodos, técnicas ou ferramentas a serem adotados para o desenvolvimento. É possível, por exemplo, implementar as exigências do modelo de referência MPS-SW com base em métodos orientados a plano (como o RUP) ou ágeis (como o Scrum). O que os modelos contêm é um conjunto de boas práticas, sem delimitar a forma como essas práticas devem ser realizadas.

Se você quiser se aprofundar em Engenharia de Software de maneira geral, recomendamos os livros de Engenharia de Software moderna de Jacobson *et al.* (2019), Sommerville (2019), Valente (2020) e Farley (2021). O livro que você tem em mãos se diferencia dos citados por ser o primeiro a enfatizar como a Engenharia de Software pode ser aplicada para sistemas inteligentes envolvendo Ciência de Dados, refletindo pesquisas de vanguarda e experiências práticas vivenciadas na iniciativa ExACTa PUC-Rio que, ao longo dos últimos anos, produziu e entregou dezenas de sistemas inteligentes para parceiros da indústria de diferentes setores de negócio, incluindo soluções patenteadas.

CAPÍTULO 2
Introdução à Ciência de Dados

Este capítulo apresentará conceitos introdutórios relacionados à Ciência de Dados e algumas definições de termos importantes. Também falaremos sobre tipos de dados e tipos de problemas de Ciência de Dados. Finalmente, apresentaremos e discutiremos o esquema básico de um projeto de Ciência de Dados.

2.1 Banco de dados ou "bando" de dados?

Nosso cenário mundial atual é caracterizado pela criação e crescimento de inúmeras bases de dados, diariamente, em fluxo contínuo e em velocidade exponencial. Em 2017, estimava-se que cerca de 90% dos dados armazenados na web tivessem sido gerados nos 2 anos anteriores. Em 2021, a internet já alcançava mais de 60% da população mundial e a quantidade total de dados gerados no mundo foi estimada em cerca de 79 *zettabytes* (79 trilhões de GB).

Esses dados são gerados, em sua maioria, de aplicativos e websites de redes sociais, bancos, *e-commerce* e sistemas internos de empresas, formando o que conhecemos como ***Big Data***, ou seja, dados de alto volume, alta velocidade e/ou alta variedade.

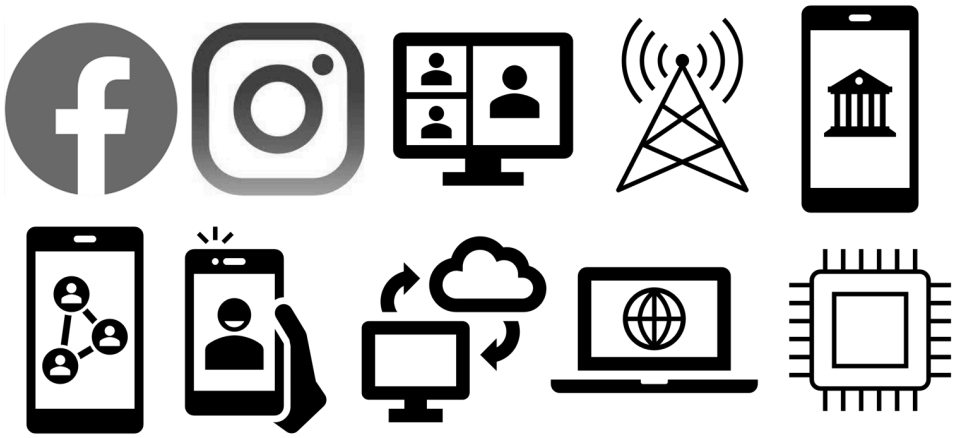


Figura 2.1: Grande variedade de dados gerados diariamente.

Para processar e obter informação útil a partir desses dados, é necessário automatizar diversas tarefas de coleta, processamento e análise de dados para tomada de decisão, uma vez que, devido ao grande volume de dados disponível, torna-se inviável realizar essas tarefas manualmente. Nesse contexto, surge a **Inteligência Artificial**, que visa simular o comportamento de um cérebro humano utilizando máquinas.

De forma mais técnica e de acordo com a definição do Gartner Group (https://www.gartner.com/en), *Data Mining* (mineração de dados) é o processo de descoberta de novas e significativas correlações, padrões e tendências em grandes volumes de dados, por meio do uso de técnicas e reconhecimento de padrões, estatística e outras ferramentas matemáticas. Para encontrar padrões, o processo de Data Mining utiliza muitas vezes as técnicas de *Machine Learning* (aprendizado de máquina), uma subárea da Inteligência Artificial que se concentra na descoberta de padrões ou de fórmulas matemáticas que expliquem o relacionamento entre os dados, e estuda formas de automatização de tarefas inteligentes que seriam difíceis de serem realizadas por humanos.

Apesar de as técnicas de reconhecimento de padrões e de análise exploratória de dados utilizadas em *Data Mining* e *Machine Learning* serem antigas e em sua grande parte provenientes da Estatística,

elas só passaram a ser efetivamente usadas para exploração de dados nos últimos anos, devido a fatores como: maior volume de dados disponível, criação e popularização de *Data Warehouses* (grandes armazéns de dados, com arquitetura de dados voltada para a tomada de decisão), recursos computacionais potentes, forte competição empresarial e criação de diversos softwares.

Por sua vez, o conceito de ***Data Science***, ou Ciência de Dados, é mais amplo: refere-se à coleta de dados de várias fontes para fins de análise, com o objetivo de apoiar a tomada de decisões, utilizando geralmente grandes quantidades de dados, de forma sistematizada. Quase sempre, além de olhar para os dados passados para entender o comportamento deles – atividade conhecida como *Business Intelligence* (BI) –, deseja-se também realizar análises de forma preditiva, por exemplo, utilizando técnicas de *Data Mining* e/ou *Machine Learning*.

Assim, Ciência de Dados não é uma ferramenta, mas um conjunto de métodos com o objetivo de apoiar decisões de negócio baseadas em dados, sendo a etapa de preparação dos dados brutos, limpeza e análise de dados fundamental para a etapa seguinte, de análise descritiva ou diagnóstica (com *Business Intelligence*) ou análise preditiva (aplicação de algoritmos de *Machine Learning*), permitindo a partir daí uma análise prescritiva. Dessa forma, é possível gerar um produto que agregue valor ao negócio, ou seja, aplicar a ciência para aprender com os dados, e não simplesmente executar os algoritmos ou técnicas de *Business Intelligence* ou *Machine Learning*.

Para trabalhar com Ciência de Dados, como o nome já indica, precisamos de dados, e quanto mais dados, melhor (desde que sejam dados com qualidade), pois será mais fácil de encontrar os padrões ou fórmulas matemáticas que os expliquem. Esses dados podem ser oriundos de fontes (estruturadas ou não) como planilhas, documentos, imagens, bancos de dados (relacionais ou não) e *Data Warehouses* e, na prática, têm qualidade ruim, sendo necessário

gastar um tempo considerável na sua preparação, limpeza e enriquecimento.

Assim, para ser capaz de realizar todas as etapas necessárias para efetivamente gerar valor ao negócio a partir de dados brutos, consideramos que o(a) profissional de Ciência de Dados deve ter uma formação multidisciplinar, unindo disciplinas como Estatística, Programação, *Machine Learning*, Banco de Dados e Inteligência Artificial.

2.2 Aplicações de Ciência de Dados

Como aplicações de Ciência de Dados já utilizadas no mundo real, podemos citar:

- No governo dos EUA, a identificação de padrões de transferência de fundos internacionais de lavagem de dinheiro do narcotráfico e prevenção de atentados;
- Na área do varejo, a utilização de dados do histórico de compras e preferências dos clientes para definir, por exemplo, a melhor organização de prateleiras. Isso pode facilitar as vendas casadas, a definição de hábitos de consumo e a previsão de necessidades de produtos/serviços em determinada região ou para determinado público-alvo, por exemplo;
- Na área da Saúde, a investigação da relação entre doenças e perfis profissionais, socioculturais, hábitos pessoais e locais de moradia, para melhor entendimento das doenças e tratamentos;
- Na área de Geologia, a identificação de litologia das rochas;
- No nosso dia a dia, também é fácil perceber diversas dessas aplicações – por exemplo, sugestão de livros na Amazon ou de filmes na Netflix; detecção automática de e-mails *spam* pelo Gmail e ligações de operadoras de cartão de crédito para confirmar transações suspeitas.

2.3 Knowledge Discovery in Databases (KDD)

É importante observar que, em problemas de Ciência de Dados, o valor dos dados armazenados está diretamente ligado à capacidade de se extrair conhecimento de alto nível a partir deles – ou seja, gerar, a partir dos dados brutos, informação útil para tomada de decisão (conhecimento). Uma das etapas que mais agrega valor é o processo conhecido como ***Knowledge Discovery in Databases*** (KDD), ou *descoberta de conhecimento em bases de dados*, que consiste em transformar dados brutos em informações úteis, gerando conhecimento para a organização. Para tal, são comumente utilizados sistemas de apoio à decisão, que auxiliam os usuários na tomada de decisão.

KDD pode ser definido como um processo de várias etapas, não trivial, interativo e iterativo, para identificação de padrões compreensíveis, válidos, novos e potencialmente úteis a partir de grandes conjuntos de dados. Essa definição mostra que a descoberta de conhecimento é um processo complexo e composto de várias etapas, que necessita do envolvimento de atores como o analista de dados (para construir modelos) e o especialista de domínio (para interpretar os resultados).

Para isso, assume-se como premissa que há um conjunto de dados (também chamado de *dataset*) que pode envolver *n* atributos (também chamados de campos, colunas, variáveis), representando o hiperespaço. Quanto maior o valor de *n* e o número de registros (também chamados de linhas, transações, exemplos, instâncias), maior o conjunto de dados a ser analisado e, geralmente, maior é a sua complexidade.

Em KDD, pode-se dividir o processo de construção dos modelos em duas grandes etapas, que são conhecidas por diversos nomes na literatura. Costuma-se dizer que se está aprendendo – ou então treinando, construindo, formulando, induzindo um modelo de conhecimento – a partir de um conjunto de dados quando se

procura por padrões nestes dados, em geral utilizando um ou mais algoritmos de *Data Mining* ou *Machine Learning*. Finalmente, quando se faz uma estimativa – ou então teste, predição – dos valores desconhecidos para atributos do conjunto de dados, diz-se que o modelo está sendo aplicado.

Suponha que temos uma base histórica de clientes e suas características como idade, renda, estado civil, valor solicitado do empréstimo e se pagou ou não o empréstimo. Esses dados são ditos rotulados, porque contêm a informação da classe a que cada cliente pertence (bom pagador/mau pagador) e são usados para a construção do modelo. A partir do momento em que novos clientes solicitarem um empréstimo, podemos utilizar o modelo construído com os dados rotulados para prever se cada novo cliente será bom ou mau pagador e, assim, apoiar a decisão do gestor. Portanto, pode-se dizer que a principal etapa do processo de KDD é o aprendizado, ou seja, a busca efetiva por conhecimentos novos e úteis a partir dos dados.

Todo processo de KDD deve ser norteado por objetivos claros. Deve-se definir previamente a tarefa a ser executada e a expectativa dos conhecedores do domínio (por exemplo, taxa de erro aceitável, necessidade de o modelo ser transparente para os especialistas, natureza das variáveis envolvidas etc.). Vale a pena ressaltar que KDD é um *processo*, e focar apenas nos resultados obtidos pode levar à desconsideração da complexidade da extração, organização e apresentação do conhecimento extraído. É um processo muito mais complexo que "simplesmente" a descoberta de padrões interessantes: envolve o entendimento do problema e estabelecimento dos objetivos, a negociação com os "donos" dos dados que se quer analisar, os recursos computacionais e profissionais disponíveis, a dificuldade de trabalhar com dados incompletos, inconsistentes e/ou incertos e a apresentação adequada dos resultados para os interessados.

2.4 Tipos de dados

A facilidade de trabalhar com dados está ligada a quanto eles são estruturados. **Dados não estruturados**, como arquivos, fotos e tweets, são mais difíceis de trabalhar do que **dados estruturados**, como planilhas e tabelas de um banco de dados. Há ainda os dados **semiestruturados**, como os e-mails, compostos por campos estruturados (como data de envio, remetente, destinatário e título) e não estruturados (como anexos e a mensagem em si).

Estima-se que pelo menos 80% dos dados do mundo sejam não estruturados, o que traz uma complexidade adicional para utilizá-los para extração de conhecimento. Os dados estruturados são mais comumente armazenados em bancos de dados relacionais e *Data Warehouses*, enquanto os dados não estruturados podem ser armazenados em *Data Lakes*.

De forma simplificada, um banco de dados relacional armazena dados em tabelas relacionadas entre si. Já um *Data Warehouse* é um repositório de dados orientado por assunto, integrado, não volátil, variável com o tempo, que favorece a extração de relatórios e análises para apoiar as decisões gerenciais. Finalmente, um *Data Lake* é um repositório único de dados brutos da empresa, que utiliza processamento e armazenamento distribuído devido ao grande volume de dados que comumente armazena. Nos bancos de dados relacionais e nos *Data Warehouses*, os dados são limpos e organizados em um único esquema antes do seu armazenamento e a análise é feita através de consultas diretamente no banco de dados (relacional ou *Data Warehouse*). Nos *Data Lakes*, os dados são armazenados em seu formato bruto e selecionados e organizados de acordo com a necessidade.

2.5 Tipos de problemas

Ciência de Dados pode englobar **análises descritivas** (O que aconteceu?), **diagnósticas** (Por que isso aconteceu?), **preditivas** (O que acontecerá?) ou **prescritivas** (O que deve ser feito?). Para as análises descritivas e diagnósticas, geralmente utilizamos técnicas de *Business Intelligence* (BI), enquanto, para a análise preditiva, podemos utilizar técnicas de *Machine Learning* (ML). Assim como para o processo de KDD, especificamente em *Machine Learning*, o *aprendizado através dos dados* é o objetivo principal. **Aprendizado** é a capacidade de se adaptar, modificar e melhorar seu comportamento e suas respostas, sendo uma das propriedades mais importantes dos seres inteligentes (humanos ou não).

Os problemas de *Machine Learning* podem ser agrupados de acordo com suas características. Os principais grandes **tipos de aprendizado**, o *supervisionado* e o *não supervisionado*, serão detalhados a seguir.

Aprendizado supervisionado

No *aprendizado supervisionado*, o modelo é construído a partir dos dados de entrada (também chamados de *dataset*), que são apresentados para um algoritmo na forma de pares ordenados (entrada – saída desejada). Dizemos que esses dados são rotulados, pois sabemos de antemão a saída esperada para cada entrada de dados. Nesse caso, o aprendizado (ou treinamento) consiste em apresentarmos para o algoritmo um número suficiente de exemplos (também chamados de registros ou instâncias) de entradas e saídas desejadas (já rotuladas previamente). Assim, o objetivo do algoritmo é aprender uma regra geral que mapeie as entradas nas saídas corretamente, o que consiste no modelo final. Os dados de entrada podem ser divididos em dois grupos:

- *X*, com os atributos (também chamados de características) a serem utilizados na determinação da classe de saída (também chamados de atributos previsores ou de predição);
- *Y*, com o atributo para o qual se deseja fazer a predição do valor de saída categórico ou numérico (também chamado de

atributo-alvo, ou *target*).

É comum que particionemos os dados de entrada (rotulados) em dois conjuntos: o **conjunto de treinamento**, que servirá para construir o modelo, e o **conjunto de teste** (também chamado na literatura de *conjunto de validação*), que servirá para verificar como o modelo se comportaria em dados não vistos, de forma que possamos ajustá-lo, se necessário, para a construção final do modelo a ser aplicado em novos dados que ainda não conhecemos a saída esperada. A figura a seguir ilustra o funcionamento do aprendizado supervisionado:

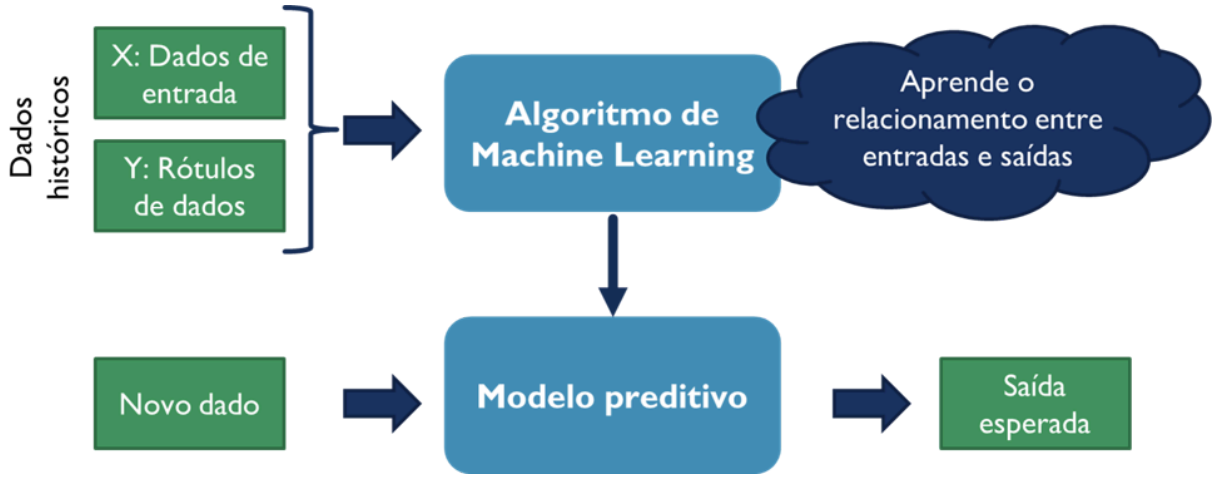


Figura 2.2: Esquema do aprendizado supervisionado.

As principais categorias de problemas de aprendizado supervisionado são **classificação** e **regressão**. Um exemplo do problema de classificação é a detecção de clientes com perfis fraudulentos. Imagine a seguinte situação: um determinado cliente deseja obter um empréstimo de R$ 1000,00. O gestor desse sistema poderia se perguntar: "Será que esse cliente vai pagar o empréstimo?", ou, ainda, "Qual é o melhor modelo de financiamento para este cliente (juros, prazo etc.)?". Esse é um problema típico de *classificação*, pois se deseja classificar um cliente em uma das possíveis classes do problema, por exemplo, bom pagador/mau pagador ou juros/prazo/outros.

Já o problema de *regressão* é parecido com o de classificação, com a diferença de que, em vez de o objetivo ser determinar uma classe, tenta-se prever um valor estimado. Um exemplo de problema de regressão é a predição do valor estimado das vendas em uma nova filial de uma determinada cadeia de lojas. Se essa questão for mapeada em um problema de classificação, as respostas possíveis poderiam ser *alto*, *médio* ou *baixo*. Se mapeada em um problema de regressão, as respostas poderiam ser valores monetários.

Aprendizado não supervisionado

No *aprendizado não supervisionado*, por sua vez, não existe a informação dos rótulos históricos, ou seja, não temos as saídas desejadas a serem estimadas e, por esse motivo, dizemos que nossos dados são *não rotulados*. Assim, o algoritmo não recebe durante o treinamento os resultados esperados, devendo descobrir por si só, por meio da exploração dos dados, os possíveis relacionamentos entre eles. Nesse caso, o processo de aprendizado busca identificar regularidades entre os dados a fim de agrupá-los ou organizá-los em função das similaridades que apresentam entre si. Como não temos dados rotulados, não há necessidade de realizar particionamento em conjuntos de treino e teste. A figura a seguir ilustra o funcionamento do aprendizado não supervisionado:

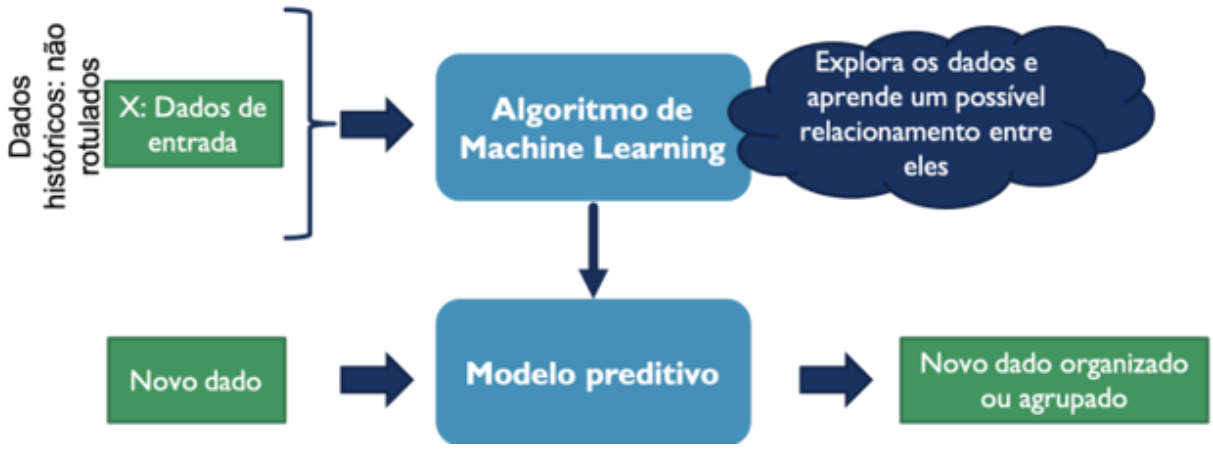


Figura 2.3: Esquema do aprendizado não supervisionado.

São exemplos de problemas de aprendizado não supervisionado a **clusterização (ou agrupamento)** e a **associação**. O problema de *clusterização* tem o objetivo de agrupar os dados de interesse. Por exemplo, se quiséssemos determinar localidades promissoras para abertura de novas filiais de uma loja, os bairros de uma cidade poderiam ser agrupados em localidades mais ou menos promissoras. Nesse caso, como não temos os rótulos de dados, é importante que os grupos formados sejam avaliados por um especialista de negócio. Já o problema de *associação* pode ser exemplificado pela oferta de novos serviços e produtos para clientes. Por exemplo, em um sistema de *e-commerce*, poderíamos nos perguntar: "Quem observa esse produto tem interesse em ver qual outro?", ou, ainda, "Quem observa esse produto costuma comprar qual outro?".

De forma análoga ao que observamos quando falamos de KDD, apesar de a essência principal de *Machine Learning* consistir na construção de algoritmos de aprendizado, essa não é a única etapa em que devemos nos concentrar. É muito importante que entendamos bem o problema a ser resolvido para que possamos traçar os objetivos principais. Em seguida, será necessário coletar e analisar os dados adequados para o problema e prepará-los, pois na maioria das vezes eles virão com informações faltantes, incompletas ou inconsistentes.

Após essas etapas, podemos construir o modelo de *Machine Learning*, que deve ser avaliado e criticado e, se necessário, voltar à etapa de coleta e análise de dados, para a obtenção de mais dados, ou mesmo retornar à etapa de construção do modelo, usando diferentes estratégias. Quando se chegar a um modelo satisfatório para o problema, será necessário apresentar os resultados para o demandante e distribuir o modelo em ambiente produtivo, não esquecendo de monitorá-lo periodicamente para verificar se continua com bom desempenho e gerando bons resultados.

2.6 Esquema básico de um projeto de Ciência de Dados

A figura a seguir ilustra o esquema básico de um projeto de Ciência de Dados, que pode ser resumido em sete etapas:

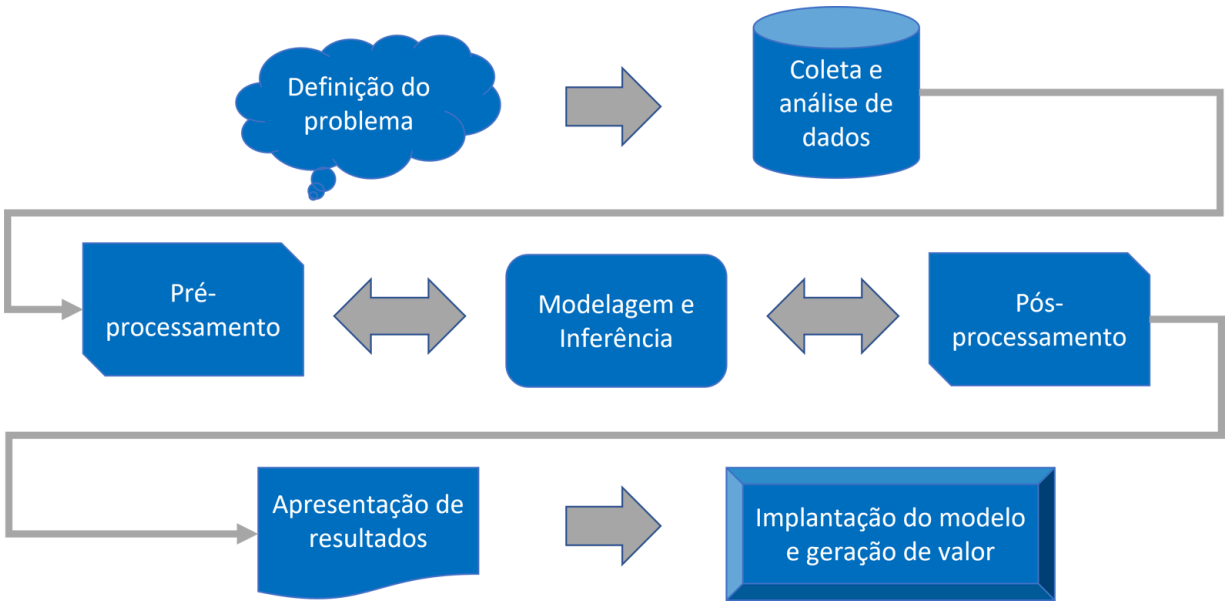


Figura 2.4: Esquema básico de um projeto de Ciência de Dados.

Um projeto de Ciência de Dados começa com uma necessidade ou ideia, compondo a etapa de **definição do problema**. Nessa etapa inicial, deve-se primeiro ter em mente o problema que se deseja resolver, e, em seguida, definir os objetivos, bem como elencar as perguntas que os gestores desejam responder.

A segunda etapa consiste em levantar as informações necessárias e efetivamente **coletar e analisar os dados** para resolver os problemas levantados na etapa anterior. Esses dados são geralmente (mas não obrigatoriamente) organizados em um ou mais bancos de dados, que podem ser bancos de dados relacionais, *Data Warehouses*, *Data Marts* ou *Data Lakes*. A partir daí, podem ser realizadas operações de ETL – *Extraction* (extração), *Transformation* (transformação), *Loading* (carga) – a partir dos dados de origem,

com a finalidade de prepará-los para os modelos que serão construídos futuramente.

A terceira etapa, na qual são realizadas as atividades de **pré-processamento dos dados**, é a mais demorada e trabalhosa de um projeto de Ciência de Dados, e estima-se que consuma pelo menos 70% do tempo total do projeto. Pode ser necessário remover ou complementar dados faltantes; corrigir ou amenizar dados discrepantes (*outliers*) e desbalanceamento entre classes, e selecionar as variáveis e instâncias mais adequadas para compor o(s) modelo(s) que serão construídos na etapa seguinte.

A quarta etapa consiste em **elencar os modelos** possíveis e passíveis para cada tipo de problema, **estimar os parâmetros** que compõem os modelos, baseando-se nas instâncias e variáveis pré-processadas na etapa anterior, e **avaliar os resultados** de cada modelo, usando métricas e um processo justo de comparação.

A seguir, na quinta etapa, são realizadas as atividades de **pós-processamento**: deve-se combinar as heurísticas de negócio com os modelos ajustados na etapa anterior e fazer uma avaliação final, tendo em vista os pontos fortes e dificuldades encontradas na implementação de cada um dos modelos.

Chega-se então à sexta etapa, a **apresentação de resultados**. Nela, recomenda-se relatar a metodologia adotada para endereçar a solução às demandas dos gestores; comparar os resultados do melhor modelo com o *benchmark* atual (caso haja) e planejar os passos para a implementação da solução de proposta.

Na sétima etapa, busca-se a **geração de valor** ao empreendimento, tanto qualitativamente (por exemplo, listar os ganhos operacionais e de recursos humanos) quanto quantitativamente (por exemplo, calculando o ROI – *Return on Investment*).

É importante ressaltar que, por mais que se fale em Ciência de Dados em temas como *Machine Learning* e Inteligência Artificial, o

papel do ser humano é fundamental na formulação dos objetivos, na escolha de técnicas de pré-processamento de dados e algoritmos utilizados, na parametrização dos algoritmos para a construção dos modelos, sempre usando a sua experiência, conhecimento e intuição para tomar as melhores decisões, bem como o método científico a fim de garantir a confiabilidade dos resultados obtidos. Além disso, soluções de Ciência de Dados continuam sendo projetos de software e devem ser construídos e mantidos seguindo os preceitos da Engenharia de Software, visando reduzir retrabalho e assegurar que esses projetos sejam entregues no prazo, dentro do custo estipulado e com a qualidade pretendida.

Parte II – Abordagens e especificação de sistemas inteligentes

Capítulo 3
Abordagens para a engenharia de sistemas inteligentes

Neste capítulo, abordaremos métodos ágeis, que são os mais utilizados pelo mercado, com considerações específicas para projetos de Ciência de Dados. Ao longo deste capítulo, daremos uma visão geral dos princípios ágeis e dos principais métodos para a gestão ágil de projetos, aplicados a projetos de Ciência de Dados. Na sequência, falaremos dos conceitos BizDev, DevOps, experimentação contínua, MLOps e DataOps, que se mostram particularmente relevantes em contextos de inovação e de soluções que envolvem sistemas inteligentes. Por fim, falaremos da abordagem *Lean R&D* (KALINOWSKI *et al.*, 2020), que temos utilizado com sucesso nesse tipo de projeto na iniciativa ExACTa PUC-Rio.

3.1 O Manifesto Ágil e princípios ágeis

O Manifesto Ágil é uma declaração de valores e princípios essenciais para o desenvolvimento de software. O documento foi criado em fevereiro de 2001, quando 17 profissionais que já praticavam métodos ágeis se reuniram no estado norte-americano de Utah. Embora esses profissionais utilizassem abordagens e métodos diferentes, eles compartilhavam dos mesmos fundamentos e decidiram escrever um documento que serviria como grito de guerra

aos novos processos de desenvolvimento de software: o **Manifesto Ágil** (BECK *et al.*, 2001). Essencialmente, esse documento serviu como base para definir o que é e o que não é desenvolvimento ágil de software. Os quatro valores fundamentais do Manifesto Ágil são:

- **Indivíduos e interações** mais que processos e ferramentas;
- **Software em funcionamento** mais que documentação abrangente;
- **Colaboração com o cliente** mais que negociação de contratos;
- **Responder a mudanças** mais que seguir um plano.

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

Contendo 12 princípios, o Manifesto Ágil tornou-se uma espécie de guia que orienta as escolhas de métodos e ferramentas de times ágeis de projetos. Segue uma descrição dos 12 princípios incluídos no Manifesto Ágil.

1. Nossa maior prioridade é **satisfazer o cliente** através da entrega contínua e adiantada de software com valor agregado;
2. **Mudanças nos requisitos são bem-vindas**, mesmo que tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente;
3. **Entregar frequentemente software funcionando**, de poucas semanas a poucos meses, com preferência à menor escala de tempo;
4. **Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto** por todo o projeto;
5. Construa projetos em torno de **indivíduos motivados**. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho;
6. O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de **conversa face a face**;

7. **Software funcionando** é a medida primária de progresso;
8. Os processos ágeis promovem **desenvolvimento sustentável**. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente;
9. Contínua atenção à **excelência técnica e bom design** aumenta a agilidade;
10. **Simplicidade**, maximizar a quantidade de trabalho não realizado é essencial;
11. As melhores arquiteturas, requisitos e designs emergem de **equipes auto-organizáveis**.
12. Em intervalos regulares, a equipe **reflete** sobre como se tornar mais eficaz e então refina e **ajusta** seu comportamento de acordo.

Cerca de 20 anos depois do Manifesto Ágil, em 2021, um conjunto internacional de pesquisadores, incluindo um dos autores deste livro, revisitou o que significa desenvolvimento ágil em uma pesquisa junto a desenvolvedores de diversos países (KUHRMANN *et al.*, 2022). Nessa pesquisa, foi possível confirmar a predominância contemporânea de desenvolvimento ágil e concluiu-se, que na percepção dos desenvolvedores, o grau de agilidade depende mais do alinhamento das práticas (gestão de *backlog*, integração contínua, desenvolvimento orientado a testes, entre outras) com os princípios ágeis do que dos métodos em si (Cascata, RUP, Scrum). Entretanto, o método predominante entre equipes ágeis é o Scrum, detalhado a seguir.

3.2 Visão geral do Scrum

Como vimos anteriormente, o Scrum é um framework de gestão ágil que utiliza tempo fixo (*sprints*) em vez de escopo fixo para determinar seus incrementos. Neste capítulo, vamos detalhar um pouco mais o Scrum, entendendo seus **pilares**, os **papéis**

envolvidos e as **cerimônias**, discutindo particularidades para o contexto de Ciência de Dados.

Pilares do Scrum

O Scrum se baseia em três pilares: **transparência**, **inspeção** e **adaptação**:

- **Transparência**: Aspectos significativos devem estar visíveis aos responsáveis pelos resultados. Ou seja, idealmente, todos os artefatos relacionados com o produto que está sendo construído e sua gestão devem estar disponíveis para toda a equipe Scrum;
- **Inspeção**: Usuários Scrum devem, frequentemente, inspecionar os artefatos Scrum e o progresso para detectar variações. Essa inspeção dos artefatos ocorre tanto ao longo das cerimônias do Scrum quanto em atividades como reuniões para o refinamento do backlog ou revisão de código.
- **Adaptação**: Se um ou mais aspectos desviou do esperado, o processo ou o material que está sendo produzido deve ser ajustado. A possibilidade constante de adaptação é a essência dos métodos ágeis. É possível adaptar, por exemplo, o que será construído ao atualizar o backlog do produto, ou, ainda, adaptar o processo ou a forma de trabalho com base na retrospectiva da *sprint*.

Note que a aplicação impulsiona uma atuação da equipe com visão crítica, capaz de lidar com mudanças e sugerir mudanças para melhorar a produtividade da equipe como um todo. Aplicando esses pilares de forma consistente, é possível dar vida a um bom processo.

Papéis no Scrum

Os papéis envolvidos no Scrum são o ***Product Owner***, o ***Scrum Master*** e a **Equipe de Desenvolvimento**.

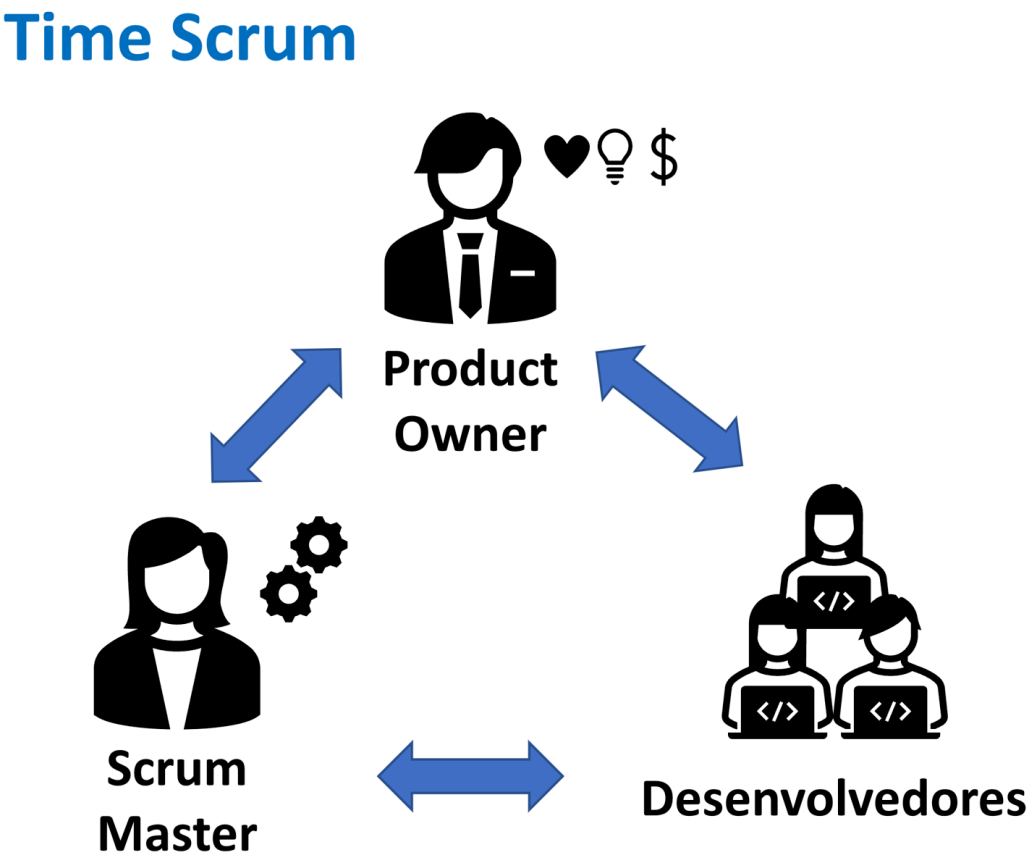


Figura 3.1: Papéis no Scrum.

O ***Product Owner*** **(PO)** foca no retorno de investimento (ROI) e no valor agregado para o cliente. Ele é o único que edita e gerencia o *backlog* do produto. A equipe deve respeitar suas decisões, ao passo que suas decisões devem sempre considerar os pontos de vista dos diferentes interessados (*stakeholders*). Conforme ilustrado na figura a seguir, além de ser responsável pelo *backlog* do produto, o PO deve buscar entender (bem) o produto, os objetivos de negócio, as necessidades dos interessados, ter visão para o produto e atuar próximo à equipe de desenvolvimento.

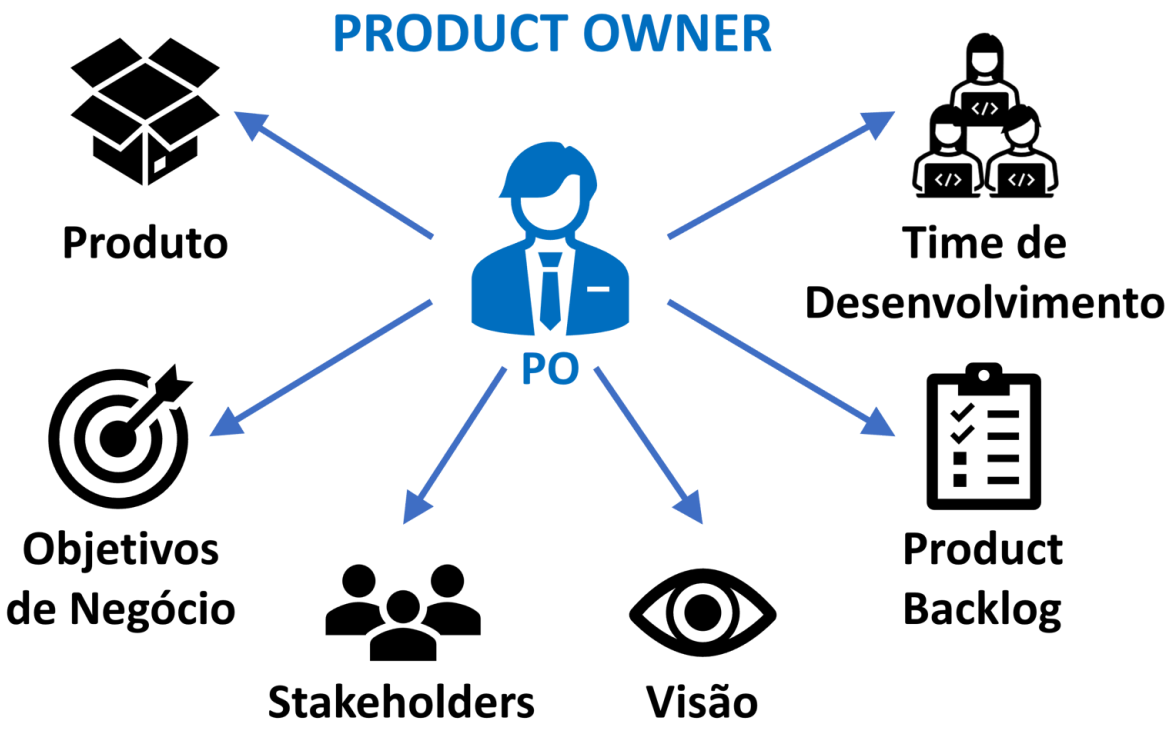


Figura 3.2: Product Owner.

O ***Scrum Master***, por sua vez, cuida do *processo*. Ele aplica o Scrum, facilita os eventos e remove impedimentos do time. É parte do seu papel ensinar a correta aplicação do processo e liderar o time rumo aos objetivos do PO. Conforme ilustrado na figura a seguir, o *Scrum Master* deve atuar como um líder servidor (que ajuda a realizar, mostra como fazer e remove impedimentos), como um *coach* e professor (que orienta e ensina sobre o processo) e como um facilitador das *cerimônias*.

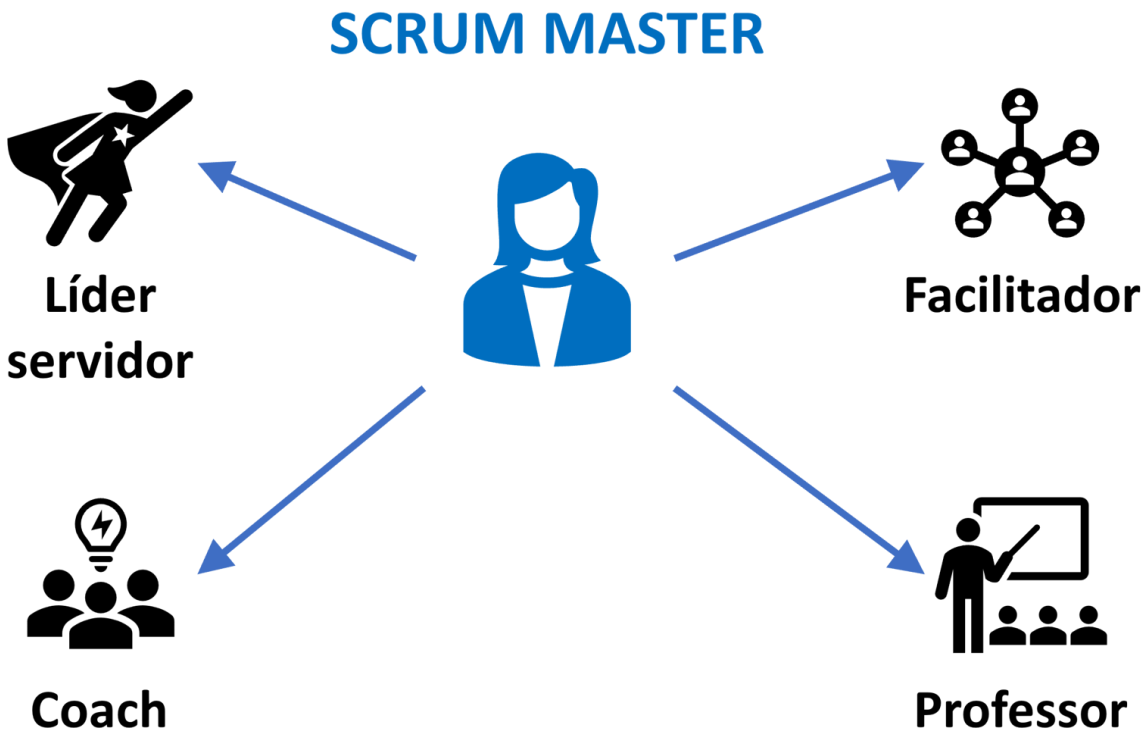


Figura 3.3: Scrum Master.

Por fim, a **equipe de desenvolvimento** deve buscar ser proativa e autogerenciável em relação ao atendimento das tarefas do *backlog* do *sprint*. A equipe deve ter atuação multifuncional (dentro do possível de sua expertise) e assumir responsabilidade compartilhada pelo desenvolvimento.

Cerimônias do Scrum

Para entender bem a dinâmica do Scrum, é fundamental entender suas **cerimônias**, que serão detalhadas a seguir.

O ***sprint*** corresponde um período fixo de tempo (normalmente de duas semanas) durante o qual um incremento utilizável do produto é criado. Para viabilizar o desenvolvimento em *sprints*, as seguintes **cerimônias** são realizadas:

1. Planejamento do *sprint*;

2. Reunião diária;
3. Desenvolvimento do *sprint*;
4. Revisão do *sprint*;
5. Retrospectiva do *sprint*.

A figura a seguir relembra a dinâmica geral do Scrum vista no capítulo 1. As cerimônias serão detalhadas na sequência.

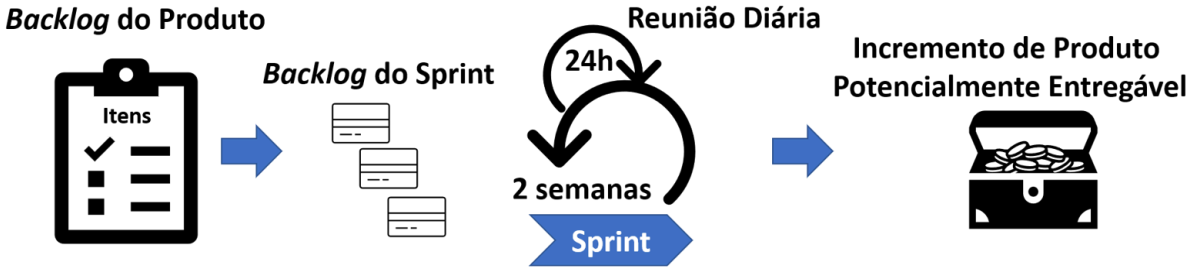


Figura 3.4: Cerimônias do Scrum.

1. Planejamento do *sprint*

Durante o **planejamento do *sprint* (*Sprint Planning*)**, o objetivo do *sprint* é definido e o trabalho a ser realizado é planejado. Isso envolve priorizar entre os itens do *backlog* do produto aqueles que devem ser desenvolvidos primeiro, analisar quais itens estão prontos para entrar no *sprint* (de acordo com a *definition of ready*) e analisar o que pode ser incluído, considerando as estimativas e a capacidade da equipe. Além da equipe Scrum, essa cerimônia conta, idealmente, com a participação de representantes do cliente. O resultado do planejamento é o *backlog* do *sprint*.

Na verdade, é comum que alguns itens do *backlog* que se revelarem prioritários, e que ainda não tenham sido detalhados ou estimados, sejam detalhados em tarefas e estimados pela equipe durante essa cerimônia, utilizando práticas como o *Planning Poker* (COHN, 2006). Entretanto, idealmente, o detalhamento e a estimativa dos itens do *backlog* prioritários são realizados previamente em sessões de refinamento do *backlog* conduzidas pelo PO, deixando o foco do

planejamento na escolha dos itens prioritários e no ajuste do que será incluído no *sprint*.

2. Desenvolvimento do *sprint*

O **desenvolvimento do *sprint* (*Sprint Execution*)** tem início assim que a equipe finaliza o planejamento dele e concorda com o conteúdo do próximo. É importante ressaltar que ninguém diz à equipe de desenvolvimento em qual ordem ou como fazer o trabalho previsto no *backlog* do *sprint*. São os membros da equipe de desenvolvimento que definem seu próprio trabalho (ao escolherem tarefas do *sprint* para realizar) e devem se auto-organizar para alcançar o objetivo do *sprint*. Todo o desenvolvimento é acompanhado também pelo *Scrum Master* e pelo PO em reuniões diárias da equipe Scrum.

3. Reunião diária

A **reunião diária (*Daily Scrum*)**, como o nome indica, envolve uma reunião diária de 15 minutos para que a equipe de desenvolvimento possa sincronizar as atividades e criar um plano para as próximas 24 horas. Idealmente, essas reuniões são realizadas sempre no mesmo horário. Durante elas, cada participante é solicitado a responder brevemente a três perguntas:

- O que eu fiz ontem?
- O que eu vou fazer hoje?
- Tenho algum impedimento?

Este é também o momento em que o *Scrum Master* assegura que o status de todas as tarefas esteja atualizado na ferramenta de gestão ágil. Para isso, normalmente se utiliza um recurso das ferramentas de gestão ágil conhecido como ***board***. O *board* mostra as tarefas do *sprint* que se encontram pendentes, em andamento e concluídas, organizadas por item do *backlog*. Para assegurar a curta duração das reuniões diárias, a equipe não deve utilizar esse momento para discutir alternativas de solução para eventuais problemas mencionados. O *Scrum Master* deverá assegurar que reuniões

separadas sejam marcadas entre os envolvidos para esse tipo de discussão.

4. Revisão do *sprint*

A **revisão do *sprint* (*Sprint Review*)** é uma atividade de inspeção e adaptação realizada no final do *sprint* para verificar o alcance dos objetivos do *sprint*. Ela envolve inspecionar e fazer o aceite dos itens do *backlog* do *sprint* que foram concluídos durante o *sprint* (de acordo com a *definition of done*). Como nem sempre todos os itens do *backlog* do *sprint* terão sido concluídos ou serão aceitos, a *sprint review* pode resultar em adaptações para o *backlog* do produto.

Assim como o planejamento, além da equipe Scrum, essa cerimônia conta idealmente com a participação de representantes do cliente. Note que a participação do cliente na revisão é um fator-chave para uma colaboração bem-sucedida. As pessoas que não estão na equipe Scrum se aproximam do que está sendo desenvolvido e ajudam a guiar sua direção. A revisão do *sprint* representa uma oportunidade agendada e fundamental para inspecionar e adaptar o produto — e isso deve estar claro para todas as partes envolvidas.

5. Retrospectiva do *sprint*

Por fim, a **retrospectiva do *sprint* (*Sprint Retrospective*)** é uma oportunidade para a equipe refletir sobre melhorias a serem aplicadas no próximo *sprint*. Essa reflexão envolve analisar o que foi bom, o que poderia melhorar e pensar em ações concretas para melhoria que poderiam ser aplicadas no próximo *sprint*.

Essas melhorias comumente envolvem formas de aperfeiçoar a comunicação para reduzir interrupções, mitigar impedimentos ou, ainda, adotar novas práticas ou ferramentas de engenharia de software, o que tem impacto direto, por exemplo, nos critérios na *definition of ready* ou *definition of done*, na forma de especificar histórias de usuário e critérios de aceitação e na revisão de código aos pares ou nos testes.

Na prática, é comum que empresas realizem as cerimônias de *revisão*, *retrospectiva* e *planejamento* em conjunto, para otimizar a agenda de interações com o cliente. Nesse caso, assim que a revisão e a retrospectiva são realizadas, inicia-se o planejamento do próximo *sprint*. Entretanto, para realizar essas três cerimônias em conjunto e garantir que haja discussão e feedback suficientes, sugerimos alocar ao menos quatro horas, sendo ao menos duas para a revisão e retrospectiva e ao menos outras duas para o planejamento — idealmente com um intervalo antes do planejamento, para que um eventual cansaço da equipe não interfira em um planejamento eficiente.

Refinamento do *backlog*

Além das cerimônias diretamente relacionadas ao *sprint*, há ainda o **refinamento do *backlog***, que envolve preparar os itens do *backlog* do produto para *sprints* futuras, refinando-os durante a *sprint* corrente para satisfazer a *definition of ready*. A equipe de desenvolvimento deveria investir de 5 a 10% do seu tempo de *sprint* para apoiar o PO em atividades de refinamento do *backlog*.

Quem conduz essa cerimônia é o PO, e ela está mais relacionada com a gestão ágil do produto do que com a gestão do projeto. Recomenda-se que o PO atue constantemente no refinamento do *backlog* e que ele conduza cerimônias específicas para tratar desse assunto junto com a equipe de desenvolvimento.

Artefatos do Scrum

Os artefatos do Scrum são o ***backlog* do produto**, o ***backlog* do sprint** e o **incremento do produto potencialmente entregável** gerado a cada nova sprint. Seguindo os pilares da *transparência*, *inspeção* e *adaptação*, eles devem estar sempre disponíveis de forma transparente para toda a equipe Scrum. Entretanto, apenas o PO deve alterar o *backlog* do produto e a descrição dos seus itens principais (por exemplo, histórias de usuário). A equipe de desenvolvimento pode ajudar a detalhar tarefas a serem realizadas

para alcançar o que está especificado nos itens definidos pelo PO. Ou seja, enquanto o PO é responsável por definir *o que+ deve ser feito, a equipe de desenvolvimento pode ajudar a definir _como* será feito.

Como vimos, o *backlog* do *sprint* representa itens detalhados e estimados do *backlog* do produto que atendem à *definition of ready* e foram incluídos nos objetivos de um *sprint*. O **incremento de produto potencialmente entregável é o resultado do *sprint***, ou seja, integra os itens acordados no *sprint* que realmente foram concluídos de acordo com a *definition of done*.

O termo *potencialmente entregável* significa que este incremento poderia ser entregue e implantado, mas que essa é uma decisão de negócios e que nem sempre o resultado do *sprint* é de fato implantado imediatamente em produção. Entretanto, esse deve ser um objetivo a ser perseguido, considerando que implantar incrementos (verificados e validados) continuamente é uma boa prática da Engenharia de Software.

Definition of ready* e *definition of done

Embora os termos *definition of ready* e *definition of done* estejam mais diretamente relacionados com a gestão ágil de produtos, eles se fazem necessariamente presentes ao explicar as cerimônias da gestão ágil de projetos com o Scrum, então vamos esclarecer esses conceitos.

O termo *definition of ready* diz respeito à verificação e validação dos requisitos, enquanto *definition of done* diz respeito à verificação e validação do software sendo construído. Vamos às definições:

***Definition of ready* (preparado)**: descreve os requisitos que devem ser atendidos para que um item possa ser movido do *backlog* do produto para o *backlog* do *sprint*.

***Definition of done* (feito)**: descreve os requisitos que devem ser atendidos para que um item possa ser considerado concluído.

A figura a seguir ilustra esses conceitos. É possível ver que atender à *definition of ready* é necessário para que um item entre no *sprint* e comece a ser feito, e que atender à *definition of done* é necessário para que ele seja considerado concluído.

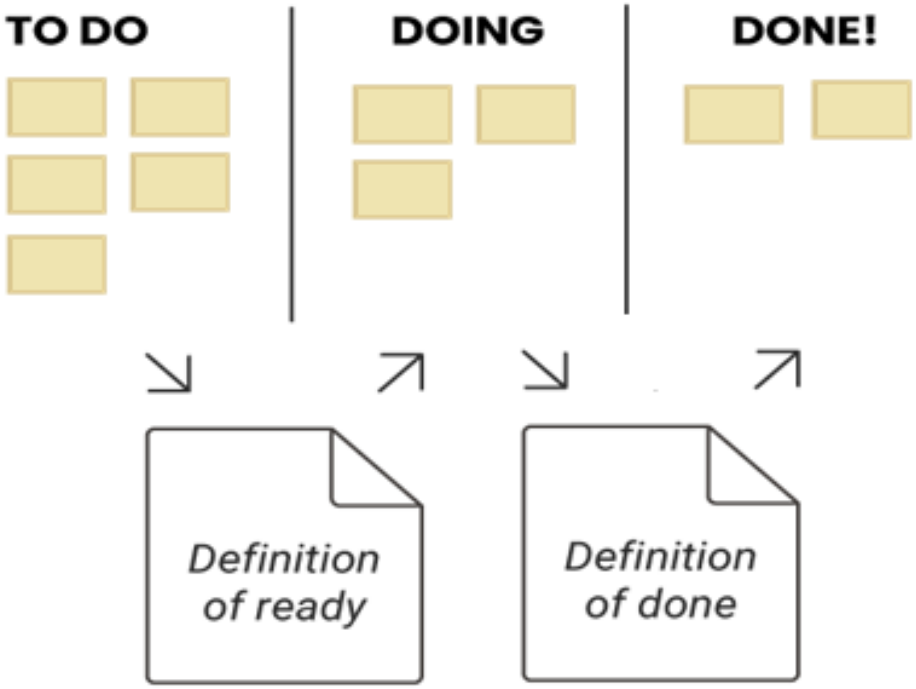


Figura 3.5: Definition of ready e definition of done.

3.3 Estimativa e velocidade

Assim como na gestão tradicional, a gestão ágil de projetos precisa lidar com a tríplice restrição de qualquer projeto: *custo*, *prazo* e *qualidade*. Ou seja, queremos entregar produtos dentro de determinado custo, no prazo estabelecido e com a qualidade esperada pelo cliente. Para isso, boas estimativas se mostram essenciais.

Iniciaremos discutindo a escolha da unidade de medida e o uso da prática de estimativa *Planning Poker*, que é bastante utilizada em contextos ágeis. Na sequência, você conhecerá o conceito de velocidade e como ele tende a se mostrar essencial para um bom planejamento.

Estimativa

Diferentemente do que muitos acreditam, o uso de métodos ágeis requer uma dedicação à estimativa, tanto em relação ao que se espera com respeito à evolução do produto como um todo ao longo do tempo quanto em relação ao que é possível de ser entregue em cada *sprint*.

Quando pensamos no produto como um todo, é comum elaborar um ***roadmap*** do produto (roteiro estratégico geral), que distribui **épicos** (conjuntos de *features* coesamente agrupadas) ou ***features*** (funcionalidades macro) a serem entregues como parte de um produto ao longo do tempo. A figura abaixo representa um exemplo de um *roadmap* do produto na ferramenta Jira, que é uma das mais utilizadas no mercado.

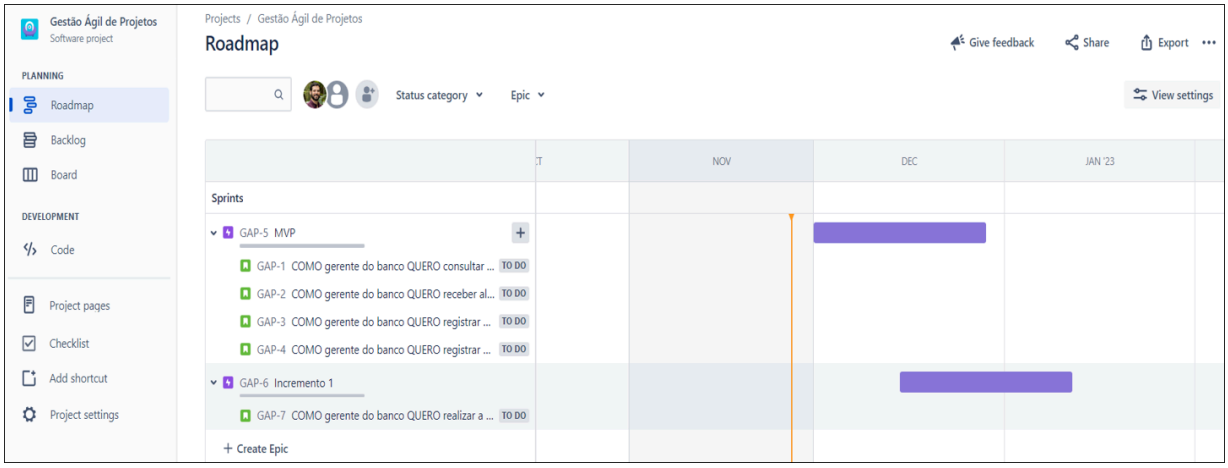


Figura 3.6: Roadmap do produto.

É possível ver na imagem que um *roadmap* se assemelha a um gráfico de Gantt da gestão de projetos tradicional. No exemplo, dois épicos estão planejados ao longo do tempo: um referente a um MVP, devendo ser iniciado e concluído em dezembro, e o outro referente a um incremento, devendo ser iniciado no meio de dezembro e concluído em janeiro. Enquanto um *roadmap* pode ser utilizado para planejar a entrega de épicos e *features* de um produto ao longo de períodos grandes (eventualmente anos), ele precisa ser constantemente atualizado e adaptado para refletir as prioridades de cada momento e um planejamento viável.

Note que cada épico possui um conjunto de histórias de usuário que fazem parte dele. A visualização dos detalhes do épico pode ser opcionalmente suprimida, já que o propósito do *roadmap* é distribuir os épicos ou *features* ao longo do tempo e não itens de granularidade menor. É possível ainda que o *roadmap* contenha épicos e *features* que ainda não foram detalhados e sobre os quais ainda se sabe pouco, mas que, de acordo com o planejamento estratégico para o produto, devem ser entregues em determinado momento.

É fundamental que o planejamento de cada *sprint* considere as prioridades definidas no *roadmap*. No exemplo da figura, o planejamento do *sprint* que se inicia em dezembro deveria priorizar

as histórias de usuário referentes ao épico do MVP, já que, de acordo com o *roadmap*, o MVP deve ser entregue ainda em dezembro.

Já em relação à estimativa do *sprint*, precisaremos de mais detalhes para entender o tamanho de cada item (por exemplo, história de usuário) e quais itens conseguimos incluir em um *sprint* de acordo com a capacidade de entrega da equipe. Essa estimativa é feita colaborativamente pela equipe de entrega do Scrum. Entretanto, antes de estimar o tamanho para cada item, é preciso definir claramente a unidade de medida para o tamanho. Embora não exista uma unidade padrão para estimar o tamanho de itens de um *backlog*, as unidades mais utilizadas por equipes ágeis são *story points* (pontos de história) e *ideal days* (dias ideais).

Ideal Days

Ideal days representam o número de pessoas-dia necessário para tratar um item do *backlog*. Note que o tempo ideal não é o mesmo que tempo decorrido. Um paralelo que facilita o entendimento é o com um jogo de futebol americano: o jogo tem quatro quartos de 15 minutos cada, mas uma partida leva até três horas e meia. Da mesma forma, um item que foi estimado para dois *ideal days* não necessariamente será entregue após dois dias corridos, já que, além de possíveis atrasos, só se consideram dias úteis de calendário e a equipe tem outras tarefas a serem realizadas. Assim, na prática, é possível que um item estimado para dois *ideal days* seja entregue após algumas semanas.

Isso pode gerar uma má interpretação na organização e até junto ao cliente. Caso na sua organização não haja o risco dessa má interpretação, é possível seguir com *ideal days*. Entretanto, o corpo de conhecimento sobre estimativas da Engenharia de Software nos orienta a entender *tamanho* e *tempo* como coisas diferentes. Note que, embora existam diversas formas de medir produtividade, há um entendimento básico de que produtividade remete a tamanho entregue (com qualidade) por tempo (SADOWSKI; ZIMMERMANN, 2019).

Quando nossa unidade de medida para tamanho remete a dias (tempo), esses conceitos naturalmente tendem a se confundir. Note que, considerando *ideal days*, entender a produtividade basicamente seria entender o tempo (ou esforço) real gasto para cada unidade de tempo (ou esforço) ideal. Fica difícil não confundir e realmente evoluir na precisão das estimativas. Os *story points* se apresentam uma alternativa bastante utilizada em métodos ágeis para estimar tamanho de itens do *backlog*.

Story points

Story points representam uma medida relativa de tamanho para itens de um *backlog*. Eles são afetados por diversos fatores, como a quantidade de funcionalidade a ser desenvolvida, a complexidade e riscos e incertezas, entre outros. Como a medida é relativa, para estimar com *story points*, é preciso estabelecer uma referência para a comparação. Uma alternativa é estabelecer como base de comparação uma história de usuário pequena do *backlog*. Algo que possa ser feito em um dia, por exemplo, incluindo o esforço de testes. Para tal, atribua um número a essa história de usuário, por exemplo, 3 *story points*. A partir daí, todas as estimativas deverão ser uma comparação relativa a essa história de usuário.

Imagine que a história de usuário de referência escolhida para valer 3 *story points* seja um cadastro CRUD básico (*create-read-update-delete*, ou seja, que permita cadastrar e manter informações) de uma entidade não complexa (por exemplo, clientes, fornecedores, equipamentos). A partir daí, é possível saber que um cadastro CRUD de algo um pouco mais complexo, como um cadastro de tipos de despesas de uma empresa que permita agrupar os diferentes tipos de despesa hierarquicamente, é maior e levaria mais *story points*, podendo ser dimensionado, por exemplo, proporcionalmente, com 5 *story points*.

Logo, a equipe precisa convencionar o que representa um *story point*. Caso sua equipe não esteja habituada com estimativas relativas ou com o uso de *story points*, recomendamos elaborar uma

matriz de *story points*, que pode servir como base para a estimativa. Isso é especialmente útil para organizações que buscam ter estimativas de tamanho padronizadas entre os seus projetos. Essa matriz pode ser customizada baseada em como o esforço, tempo, complexidade e risco afetam a percepção de tamanho pela equipe. A figura a seguir representa um exemplo para esse tipo de matriz.

Story Points	Esforço	Tempo	Complexidade	Risco e Incerteza
1	Mínimo	Poucos Minutos	Pequena	Nenhum
2	Mínimo	Algumas Horas	Pequena	Nenhum
3	Leve	Um Dia	Baixa	Baixo
5	Moderado	Alguns Dias	Média	Moderado
8	Alto	Uma Semana	Média	Moderado
13	Muito Alto	Certamente mais que uma semana	Alta	Alto

Figura 3.7: Matriz de story points.

Planning Poker

O *Planning Poker* é uma técnica baseada em consenso para a estimativa de esforço com base na opinião de especialistas. Toda a equipe Scrum participa e debate para expor suposições, adquirir entendimento compartilhado e dimensionar o item em questão. A equipe faz uso do seu histórico de estimativas para estimar mais facilmente o próximo conjunto de itens, gerando estimativas relativas.

O *Planning Poker* faz uso de cartas que seguem uma escala para a estimativa de *story points*. A escala mais frequentemente usada é a popularizada por Mike Cohn através do livro *Agile Estimating and Planning* (2006). Essa escala é baseada na sequência de Fibonacci e está ilustrada na figura a seguir.

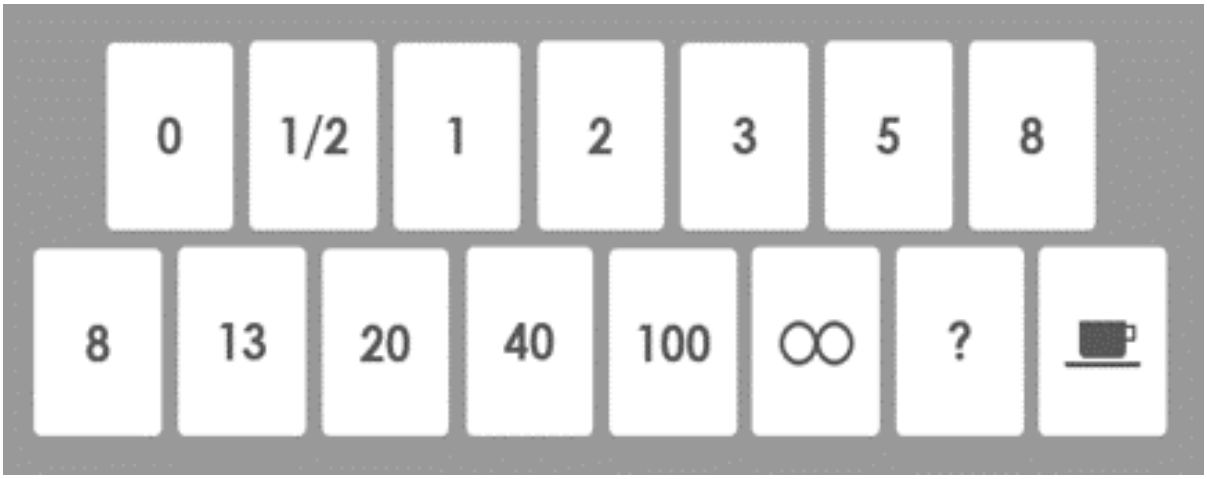


Figura 3.8: Cartas do Planning Poker.

Veja a seguir o significado de cada uma das cartas dessa escala:

- `0` — Usada quando o item já foi feito ou é tão pequeno que não faz sentido dar um número.
- `1/2` — Usada quando o item já foi feito ou é tão pequeno que não faz sentido dar um número
- `1`, `2` e `3` — Usadas para dimensionar itens pequenos.
- `5`, `8` e `13` — Usadas para dimensionar itens médios. É comum que equipes tentem quebrar itens maiores que 13 em um conjunto de itens menores.
- `20` e `40` — Usadas para dimensionar itens grandes (por exemplo, *features* inteiras).
- `100` — Usada para indicar que o item é uma *feature* muito grande ou um épico.
- *Infinito* — Usada para indicar que um item é tão grande que não faz nem sentido colocar um número nele.
- `?` — Usada para indicar que o item não foi suficientemente compreendido e solicitar explicações adicionais ao PO.

- *Café* — Usada para solicitar um intervalo na discussão. Se as pessoas estiverem jogando essa carta, a equipe precisa de uma pausa.

Após distribuir as cartas para os participantes (a equipe de desenvolvimento), o *Planning Poker* é realizado seguindo estes sete passos:

1. O PO seleciona um item do *backlog* (em geral, os itens são histórias de usuário) e lê para a equipe;
2. Os membros da equipe discutem o item;
3. Cada membro seleciona uma carta, sem revelá-la aos demais;
4. As estimativas são expostas simultaneamente;
5. Caso haja consenso, a estimativa foi alcançada;
6. Se não há consenso, as estimativas individuais são discutidas;
7. Cada membro seleciona novamente uma carta, até que o consenso seja alcançado.

No caso de divergências nas estimativas, a discussão do passo 6 deverá ser liderada pelo PO, que normalmente inicia a discussão perguntando as razões para as estimativas mais altas e mais baixas apresentadas e abre para outras considerações. Com base nos esclarecimentos, a equipe tende a convergir e o consenso é usualmente alcançado após duas ou três rodadas.

É possível aplicar a técnica tanto presencialmente quanto remotamente com o apoio de uma ferramenta web. A ferramenta *Planning Poker Online*, por exemplo, permite importar *issues* do Jira e aplicar estimativas em equipe. Essa ferramenta, aliada a uma ferramenta tradicional de videoconferência, como o *Zoom* ou o *Teams*, permite a aplicação remota da técnica sem maiores dificuldades.

Velocidade

Velocidade é a quantidade de trabalho que a equipe consegue completar em um *sprint*. Ela é obtida de forma simples somando os

tamanhos (por exemplo, *story points*) de todos os itens do *backlog* que foram completados ao final do *sprint*. É importante deixar claro que apenas itens completados (que atendem à *definition of done*) devem ter seus tamanhos somados, já que em projetos ágeis o progresso deve ser medido estritamente pelo que é efetivamente entregue para o cliente.

Conhecer a velocidade média de uma equipe permite estimar o número de *sprints* necessários para uma entrega. Além disso, permite realizar um planejamento de *sprint* mais consciente, de acordo com a capacidade efetiva da equipe. Não deveríamos incluir em um *sprint*, por exemplo, itens cuja soma de tamanho ultrapasse a velocidade média da equipe alocada. Note que, além da velocidade, a determinação da capacidade também deve considerar a disponibilidade dos profissionais envolvidos para o projeto em questão durante o *sprint*.

Conhecer a velocidade média da equipe permite ainda acompanhar o progresso do *sprint* através de gráficos de *burndown*, que podem ser utilizados para monitorar o trabalho remanescente ao longo do tempo. No dia zero do *sprint*, essa quantidade corresponde ao tamanho de tudo que foi incluído no *sprint*, o que, em geral, corresponde aproximadamente à velocidade média da equipe. À medida que os itens do *backlog* vão sendo concluídos, o tamanho desses itens vai sendo descontado do trabalho remanescente.

A figura a seguir exibe um gráfico de *burndown* do Jira. A linha mais uniforme indica o *burndown* ideal com o progresso esperado (*guideline*) para zerar o trabalho remanescente no *sprint*; a outra linha, com reduções mais abruptas à medida que histórias de usuário vão sendo entregues, indica o real trabalho remanescente.

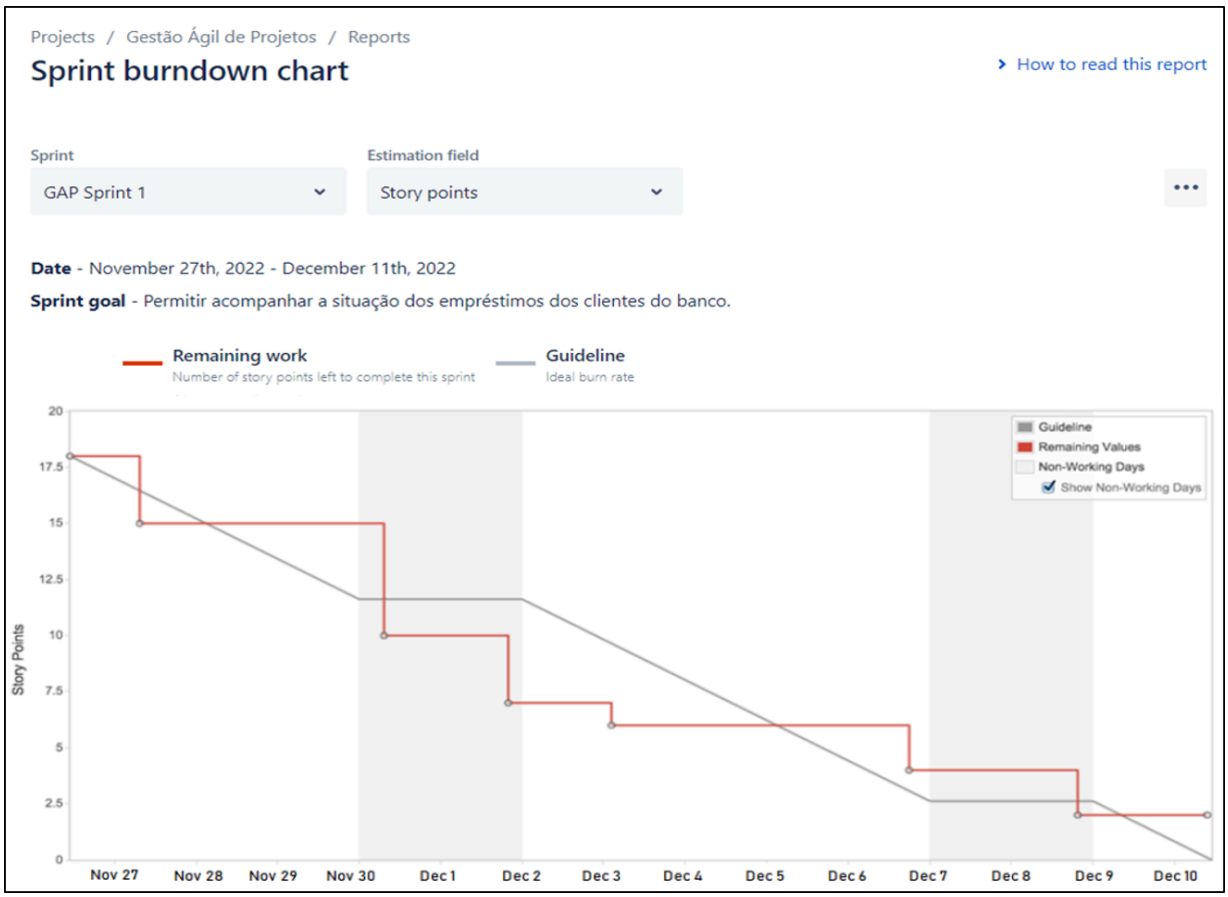


Figura 3.9: Gráfico de burndown da ferramenta Jira indicando a guideline e o trabalho remanescente.

O cálculo da velocidade média de uma equipe presume que a equipe já esteja atuando em conjunto e que se tenha dados históricos a respeito dos tamanhos totais entregues pela equipe nos últimos *sprints*. No caso de não haver dados históricos, o ponto de partida inicial para a velocidade deve ser a soma do tamanho resultante de um planejamento mais cauteloso do *sprint* pela equipe. Esse número deve então ser ajustado à medida que a equipe realiza *sprints* e começa a obter dados históricos.

3.4 Scrum em projetos de Ciência de Dados

Nesta seção, faremos algumas considerações sobre adaptações para a aplicação do Scrum em projetos de sistemas inteligentes, que envolvem componentes de *Machine Learning*. As adaptações aqui discutidas são baseadas em lições aprendidas em projetos reais com diversas retrospectivas.

Um aspecto fundamental para o uso do Scrum em projetos de Ciência de Dados é que o PO entenda bem sobre requisitos para sistemas inteligentes, conhecendo perspectivas e preocupações que devem ser consideradas na especificação desse tipo de sistema, como as apresentadas na técnica PerSpecML (VILLAMIZAR *et al.*, 2023), que será detalhada no capítulo seguinte. Essa visão geral é essencial para construir um sistema inteligente que efetivamente atenda às necessidades dos clientes.

Como você verá ao estudar sobre requisitos de sistemas inteligentes no capítulo seguinte, uma *feature* relacionada com uma inferência de *Machine Learning* normalmente se traduz em uma única história de usuário. Por exemplo:

FEATURE:

(F01) Classificação automática na concessão de empréstimo.

User Story:

(US01) **Como** gerente do banco, **Quero** realizar a classificação automática de um cliente **Para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão.

Acaba sendo somente uma história de usuário porque o usuário só perceberá valor agregado quando puder solicitar a classificação automática e puder visualizar o seu resultado. Entretanto, essa especificação é claramente insuficiente para subsidiar o desenvolvimento. Como vimos no capítulo 2 sobre Ciência de Dados, para realizar uma classificação automática, tipicamente será

necessário coletar e integrar dados (possivelmente de diversas fontes), realizar uma análise exploratória, pré-processar os dados, construir e avaliar o modelo, implantar o modelo e construir o front-end que exibirá o resultado da classificação.

Ou seja, essa história de usuário provavelmente vai ultrapassar a duração de um *sprint* para ser finalizada. Para piorar, no Scrum, o progresso é medido pela funcionalidade entregue para o usuário, ou seja, ela só somará para o progresso do projeto (normalmente com o número de *story points* atribuído à história de usuário) quando tudo isso tiver sido realizado, o que, dependendo do contexto, poderá levar meses de trabalho.

Mesmo que os desenvolvedores realizem a quebra da história de usuário em diversas tarefas mais específicas, no Scrum a finalização dessas tarefas não é contabilizada para o progresso do *sprint*, já que o usuário tipicamente só terá valor agregado quando toda a história de usuário estiver concluída. Uma alternativa de solução é quebrar a história de usuário em partes relacionadas com as tarefas a serem realizadas. Por exemplo:

(US01) **Como** gerente do banco, **Quero** realizar a classificação automática de um cliente **Para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão — Parte I – Coleta e integração;

(US01) **Como** Gerente do Banco, **Quero** realizar a classificação automática de um cliente **Para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão — Parte II – Análise exploratória e pré-processamento;

(US01) **Como** gerente do banco, **Quero** realizar a classificação automática de um cliente **Para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão — Parte III – Construção e avaliação do modelo;

(US01) **Como** Gerente do Banco, **Quero** realizar a classificação automática de um cliente **Para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão — Parte IV – Integração com o *front-end*.

Embora isso fira a ideia do Scrum de que cada história de usuário deveria permitir ao usuário realizar algo que ele não era capaz de realizar antes, isso facilita a gestão do progresso, evitando que se fique com a mesma história de usuário sendo propagada entre os *sprints*. Afinal, dessa forma é possível mensurar o tamanho (por exemplo, em *story points*) de cada uma das partes da história de usuário e medir o progresso dessa história de usuário de forma parcial.

Na prática, embora esse tipo de quebra não esteja prevista no Scrum, isso acaba se mostrando necessário, considerando que uma história de usuário que envolve *Machine Learning* é naturalmente mais complexa do que histórias de usuário que tratam do processamento de transações convencionais.

Outra diferença é que, enquanto requisitos não funcionais transversais ao sistema (como aspectos de segurança ou usabilidade a serem considerados para todo o sistema) tendem a ser especificados na *definition of done* para assegurar sua verificação para todas as histórias de usuário, histórias referentes a *Machine Learning* tendem a ter requisitos não funcionais específicos (relacionadas com confiabilidade ou explicabilidade, por exemplo, que devem ser detalhados para a história de usuário e incluídos em seus critérios de aceitação específicos.

Embora de fato o Scrum requeira algumas adaptações quando comparado ao desenvolvimento de sistemas de processamento de transações convencionais, como as citadas acima, nos projetos que temos vivenciado ele tem se mostrado adequado para a construção ágil de sistemas inteligentes.

3.5 Produção just-in-time — Lean e Kanban

A produção *just-in-time* procura adequar a produção à demanda, fornecendo apenas o que efetivamente encomendado e focando na eficiência, produtividade e redução de desperdícios. Dois métodos ágeis bastante discutidos na atualidade e relacionados com esse conceito são o Lean e o Kanban.

Lean

Lean é um método de produção que visa reduzir continuamente os tempos de produção e de resposta de fornecedores e clientes por meio de uma produção "enxuta" (*lean*), que foca na entrega de valor da perspectiva do cliente e na eliminação do desperdício. Lean está fortemente relacionado com o modelo operacional implementado nas décadas de 1950 e 1960 pela empresa automobilística Toyota, chamado de Sistema Toyota de Produção (OHNO, 1988), que se

baseia no gerenciamento de estoque *just-in-time* e no controle de qualidade automatizado.

Lean adota a abordagem *just-in-time* e busca a entrega de valor (da perspectiva do cliente), a eliminação do desperdício (com o que não agrega valor para o cliente) e a melhoria contínua. A adaptação do Lean para o contexto de desenvolvimento de software ocorreu com o livro *Lean Software Development*, escrito por Poppendieck e Poppendieck (2003). Essencialmente, o livro reafirma os princípios lean tradicionais e relaciona o que se prega no Lean com práticas ágeis. De fato, o desenvolvimento de software lean é considerado um método ágil.

O desenvolvimento de software lean pode ser resumido em sete princípios, muito próximos aos princípios da produção lean tradicional (POPPENDIECK; POPPENDIECK, 2003):

1. **Eliminar desperdício.** Se alguma atividade puder ser contornada ou o resultado puder ser alcançado sem ela, ela é considerada *desperdício*. Para eliminar o desperdício, deve-se entender como ele se manifesta e ser capaz de identificá-lo. Esse é um dos princípios fundamentais do Lean e falaremos mais dele a seguir;

2. **Ampliar o aprendizado.** O conhecimento adquirido deve ser compartilhado. No desenvolvimento de software, o processo de aprendizado pode ser acelerado pelo uso de ciclos de iteração curtos (como *sprints*) que contemplem o *feedback* de clientes (como revisões de *sprint*) e reflexões para o aprendizado e adaptação (como retrospectivas de *sprint*).

3. **Adiar as decisões.** Realize testes e experimentos antes de tomar decisões irreversíveis. Gerencie requisitos pensando no *backlog* do seu produto como algo que pode mudar. Apenas se comprometa com decisões e detalhamentos para um período próximo e previsível, deixando o restante em aberto e adiando as demais decisões, entendendo que mudanças podem ocorrer.

4. **Entregar o mais rápido possível.** Funcionalidades implementadas devem ser entregues o quanto antes. Utilize o *feedback* dos clientes como um termômetro para discutir melhorias. Se o produto for falhar, isso deve ocorrer de forma rápida e servir para gerar aprendizado.

5. **Empoderar a equipe.** Lean segue o princípio ágil que vimos anteriormente de construir projetos em torno de indivíduos motivados, de dar a eles o ambiente e o suporte necessário e confiar neles para fazer o trabalho. Ou seja, em vez de microgerenciar e cobrar o progresso das pessoas, o progresso deve ser incentivado e a equipe deve se autogerenciar. Um ambiente de respeito mútuo, com uma comunicação proativa e *feedback* constante tende a gerar uma atmosfera produtiva e colaborativa.

6. **Construir com qualidade.** Técnicas que permitam o controle da qualidade do produto devem ser incorporadas ao processo. Essas técnicas variam de acordo com o tipo de software e sua criticidade, mas podem incluir, por exemplo, análise estática, revisões de código modernas, diferentes tipos de teste e análise causal de defeitos.

7. **Otimizar.** O processo deve ser continuamente melhorado, tornando-o o mais eficiente possível. Um caminho promissor para otimizar o processo é evitar a recorrência de problemas através da análise causal (KALINOWSKI *et al.*, 2012). Outra forma de otimização é buscar a integração consciente de inovações no processo, o que pode ser impulsionado através de parcerias com grupos de pesquisa focados em Engenharia de Software.

Acredita-se que quando esses sete princípios estão implementados em conjunto e são bem compreendidos por todos os participantes de um projeto, há uma boa base para o desenvolvimento de software com sucesso. Embora todos os princípios sejam relevantes, a dedicação implacável à eliminação do desperdício é, sem dúvida,

uma das ideias-chave do Lean. Vamos, então, entender um pouco mais sobre como o desperdício se manifesta em projetos de desenvolvimento de software.

Eliminação de desperdício em desenvolvimento de software

Para eliminar o desperdício, é preciso ser capaz de identificá-lo. Tudo que não agrega valor ao produto deve ser eliminado. A seguir, são listados os oito tipos de desperdício apresentados em *Lean Software Development* (POPPENDIECK; POPPENDIECK, 2013), com exemplos de como esses desperdícios se manifestam em projetos de desenvolvimento de software.

1. **Trabalho parcialmente feito.** No Lean, o trabalho parcialmente feito contempla: *features* especificadas muito antes do seu desenvolvimento; código não integrado e implantado; código não testado e código não documentado — além da famosa *dívida técnica*, que representa trabalho feito de forma incompleta ou não da melhor forma possível. O primeiro item, o das features especificadas muito antes do seu desenvolvimento, pode causar estranheza, mas é isso mesmo. No contexto Lean, um *backlog* detalhando *feature* para muitas *sprints* à frente deve ser evitado. Lembre-se de que, no Lean, temos a produção alinhada com a demanda (*just-in-time*).

2. ***Features* extras.** Contemplam tudo o que foi desenvolvido e que acaba não sendo usado ou que não agrega valor para o cliente. Essas *features* extras devem ser evitadas, mantendo o produto "enxuto", independente de elas terem surgido pela vontade dos desenvolvedores de entregar além do que foi pedido (fenômeno conhecido como *Gold Plating*, em português, *folheamento a ouro*), terem sido defendidas por um dos interessados no produto, ou até terem sido definidas em consenso entre o cliente e os desenvolvedores. Se não está sendo usada ou não está agregando valor, não deveria ter sido feita.

3. **Reaprendizado.** Representa o tempo gasto para reaprender algo que já se sabia. Em desenvolvimento de software há vários exemplos de pessoas investindo tempo para reaprender algo que já se soube na empresa. Mudanças frequentes de tarefas e código mal-escrito ou maldocumentado, por exemplo, levam a ter que frequentemente reaprender como um código funciona. Não capturar adequadamente conhecimento também pode levar à necessidade de redescoberta desse conhecimento. Pense, por exemplo, em uma regra de negócios complexa e relevante, que foi compreendida pelo PO, mas não documentada. Caso esse conhecimento seja esquecido, ele terá que ser novamente obtido. Todo o esforço investido em descobrir o que já se sabia é desperdício.

4. **Alternar tarefas (*tasks switching*).** Desenvolvedores de software tipicamente requerem profunda concentração para realizar bem o seu trabalho. Interrupções são o inimigo da profunda concentração, e mudança de tarefas é uma interrupção — isso é fundamental em desenvolvimento de software. DeMarco e Lister (2013) descrevem que estados de desenvolvimento de altíssima produtividade (que denominam de *flow*) requerem algum tempo sem interrupções para que sejam alcançados e mantidos.

5. **Atrasos.** Exemplos de atrasos em desenvolvimento de software incluem: demora para iniciar o projeto; gargalos de muito trabalho em andamento (WIP – *work in progress*) em determinada fase do processo, refletindo na frequente espera da conclusão de uma atividade anterior; demora para iniciar uma atividade após a conclusão de sua predecessora; atividades de aprovação entre fases de desenvolvimento que requerem a atenção de pessoas pouco disponíveis; e demoras para a integração e implantação após a conclusão de uma atividade.

6. **Passagem de trabalho (*handoff*)**. Uma transferência ocorre sempre que um trabalho é passado de uma pessoa para outra. Cada vez que isso ocorre, deixamos para trás parte do

conhecimento que gostaríamos de transferir. Por isso, devemos documentar bem o conhecimento necessário para agregar valor para o cliente e reduzir o número dessas transferências ao máximo.

7. **Defeitos.** Defeitos podem ser introduzidos em diferentes momentos e em diferentes artefatos. Um defeito pode estar, por exemplo, em uma história de usuário especificada de forma incorreta ou ambígua, em uma regra de negócio ou em um trecho de código. Os defeitos mais caros para serem corrigidos são os introduzidos no início do projeto (por exemplo, nas especificações por compreender equivocadamente uma necessidade do cliente) e só detectados com o software em uso. Por isso, eles devem ser detectados assim que possível — idealmente, não deveriam nem ser introduzidos. A detecção rápida dos defeitos envolve incorporar no processo atividades de controle da qualidade, como revisões e testes. Evitar sua introdução, por sua vez, pode ser alcançado de forma eficaz otimizando o processo através da uma análise causal de defeitos (KALINOWSKI *et al.*, 2012).

8. **Sobrecarga gerencial.** Atividades gerenciais que não ajudam a agregar valor para o cliente são consideradas desperdício e deveriam ser simplificadas ou eliminadas. Muitas vezes, processos burocráticos exigem documentos de gestão como planos de projeto enormes (comumente cheios de informações essencialmente repetidas de outros projetos); relatórios de progresso com informações redundantes ou que poderiam ser obtidas de forma automatizada; atas desnecessárias e outros, que nem sempre estão objetivamente alinhados com a entrega de valor e que acabam correndo em paralelo , em vez de afetar a eficiência da entrega de valor.

Ou seja, se alguma atividade puder ser contornada ou o resultado puder ser alcançado sem ela, ela é considerada *desperdício*, devendo ser identificada e eliminada. Trabalho parcialmente feito e eventualmente abandonado durante o desenvolvimento é

desperdício. *Features* extras que não são usadas com frequência pelos clientes são desperdício. Reaprendizado de requisitos para concluir o trabalho é desperdício. Alternar tarefas das pessoas é desperdício devido ao tempo gasto na troca de contexto. Atrasos em qualquer parte do desenvolvimento representam desperdício. Passagem de trabalho demasiada de uma pessoa para outra é desperdício. Defeitos e falta de qualidade representam desperdício. Sobrecarga gerencial que não agrega valor é desperdício.

Esperamos que, agora, conhecendo os princípios do Lean e os desperdícios típicos a serem eliminados do desenvolvimento de software, independentemente da abordagem específica que você utiliza, você "leve o lixo para fora" com frequência, mantendo seu processo enxuto (*lean*).

Em geral, as empresas que utilizam Lean no desenvolvimento de software fazem isso como uma estratégia para aumentar a eficiência, assegurando que só se gere o que foi efetivamente demandado ao longo do processo de produção, reduzindo o estoque de trabalho parcialmente feito, custos e desperdício. Após definir o fluxo de valor, mapeando as atividades que agregam valor ao produto, utilizam o conceito de produção *just-in-time* e controlam o fluxo de produção (desenvolvimento) com base nos princípios do Lean. Uma forma visualmente eficaz e bastante utilizada para controlar o fluxo de produção seguindo o conceito de produção *just-in-time* é utilizar o Kanban.

Kanban

O **Kanban** permite controlar o fluxo de produção de forma visual e intuitiva, com foco na eficiência. No Kanban, cartões visuais são usados para controlar o fluxo de produção utilizando o conceito de produção *just-in-time*, em que as tarefas são preparadas sob demanda para as atividades seguintes, caracterizando um sistema de produção baseado em *pull*. De fato, a palavra japonesa *kanban* significa "cartão visual". Um aspecto importante é que as tarefas devem ser preparadas e então "puxadas" pela equipe responsável

pela atividade seguinte, e não "empurradas" para a equipe da atividade seguinte.

Alinhado aos pilares do Lean, o Kanban busca ajudar times de desenvolvimento a trabalhar em um ritmo sustentável, entregando valor para o cliente, eliminando desperdício e focando na melhoria contínua (ANDERSON, 2010). Em muitas empresas o Kanban é utilizado como uma alternativa para o Scrum. Comparado ao Scrum, o Kanban é mais simples, não estabelece *sprints* nem papéis e cerimônias. O único artefato do Kanban é o *quadro Kanban*. Vamos entender como esse quadro pode ser utilizado para controlar um fluxo de produção *just-in-time* em projetos de desenvolvimento de software.

Quadro Kanban

O **quadro Kanban** é dividido em colunas. A primeira coluna representa o *backlog* do produto. No contexto de desenvolvimento de software, cada cartão tipicamente representa uma história de usuário — ou seja, uma funcionalidade que vai permitir a algum usuário realizar algo novo no sistema. As demais colunas são atividades que representam o fluxo de valor, ou seja, para transformar histórias do usuário em funcionalidades que agreguem valor para o cliente. Cada uma dessas colunas é dividida em duas, uma representando os cartões em andamento e outra representando os cartões concluídos. Essencialmente, os cartões concluídos em uma atividade aguardam para ser puxados (*pull*) por um membro da equipe para a atividade seguinte.

O Kanban presume equipes **autogerenciadas** e **multifuncionais** — *autogerenciadas* porque cada membro tem autonomia para definir qual cartão vai ser puxado para a próxima atividade; e *multifuncionais* porque a equipe deve incluir membros capazes de realizar todas as atividades que fazem parte do fluxo de valor.

Há uma regra a ser obedecida por toda a equipe para puxar cartões de uma atividade para outra: respeitar os **limites WIP** (*Work in*

Progress). Para evitar sobrecarga de trabalho em determinada atividade e a formação de gargalos que possam impedir o progresso do projeto como um todo, os *limites WIP* estabelecem o número limite de cartões que podem estar em cada atividade do quadro Kanban.

Esses limites consideram tanto os cartões na primeira coluna (em andamento) quanto os na segunda coluna (concluídos) e se aplicam a todas as colunas, exceto a primeira (*backlog*) e a última atividade do fluxo de valor, em que só a coluna em andamento deve ser considerada, já que a conclusão dessa atividade representa entrega e não gera gargalo.

A figura a seguir representa um quadro Kanban que adaptamos para este capítulo a partir do *template* Kanban da ferramenta Miro. É possível ver que não há limites WIP para o *backlog* nem para a coluna de cartões concluídos da última atividade — que neste exemplo é a verificação da *definition of done*. Eles representam os dois extremos do que se visualiza em relação ao que deve ser feito e ao que já foi feito.

Backlog	**Especificação** WIP: 4		**Implementação** WIP: 6		**Verificação de *definition of done*** WIP: 2	
Backlog |6	**Em especificação | 2**	**Especificadas | 2**	**Em implementação| 3**	**Implementadas | 3**	**Em verificação | 1**	**Verificadas | 3**
US15	**US13**	**US11**	**US08**	**US05**	**US03**	**US01**
US16	**US14**	**US12**	**US09**	**US06**		**US02**
US17			**US10**	**US07**		**US04**
US18						
US19						
US20						

Figura 3.10: Exemplo de quadro Kanban com histórias de usuário e limites WIP.

Vamos explicar a produção *just-in-time* com base nessa figura. Para iniciar a especificação de uma nova história de usuário, antes será preciso que alguma das quatro atualmente sendo especificadas seja

puxada pela equipe para a implementação. Isso evitará a formação de um estoque de histórias especificadas que a equipe não será capaz de tratar. Por sua vez, para que a equipe puxe uma história especificada para a implementação, será necessário primeiro que mais uma história seja puxada para a verificação da *definition of done*.

É possível ver que há espaço para essa verificação na última coluna, em que o limite WIP é 2, mas em que apenas uma história está sendo verificada neste momento. Assim, no exemplo, a equipe deverá se concentrar em primeiro puxar uma história com implementação concluída da coluna de implementação para a coluna de verificação do DoD, para a partir daí poder puxar uma história da coluna de especificação para a coluna de implementação. Isso, por sua vez, abrirá espaço para a especificação de mais uma história do *backlog*, já que a implementação está dando andamento ao que já estava especificado até então e demandando mais especificação. De maneira geral, isso ajuda a alinhar a produção da equipe como um todo à demanda, produzindo em um ritmo sustentável e alinhando a produção entre as atividades, evitando a formação de gargalos.

Uma ressalva é que embora as histórias de usuário sejam comumente quebradas em tarefas, com base em vivências práticas e seguindo os preceitos do Lean de forma rigorosa, recomendo que se utilize o quadro Kanban com histórias de usuário e não com as tarefas. Note que o quadro Kanban deve representar o fluxo de valor e que o usuário só perceberá valor quando uma nova história de usuário for entregue, possibilitando a ele fazer algo no sistema do qual ele não era capaz antes. A história de usuário acaba sendo a unidade de insumo para as últimas tarefas referentes à entrega do valor, como a verificação da *definition of done (DoD)*.

Para que isso funcione bem, é preciso assegurar que as histórias de usuário estejam na granularidade adequada. Esse trabalho de ajuste e quebra de histórias de usuário é normalmente realizado pelo *Product Owner* no contexto do refinamento do *backlog*. Claro que nada impede que internamente haja um controle das tarefas que

estão sendo realizadas para a implementação de uma história de usuário, por exemplo, através de um *issue tracker* (como o Jira). Os *issue trackers* podem ser facilmente configurados para ter um *board* no formato do quadro Kanban focado no progresso das histórias de usuário e no fluxo de valor, e outro para monitorar o progresso da tarefas de implementação das *user stories*.

Uma alternativa seria quebrar a história de usuário em tarefas no próprio quadro Kanban, na atividade de especificação e a partir daí monitorar o andamento de tarefas. Além de o quadro nesse caso não representar o fluxo de valor, surge uma dificuldade prática quando essa abordagem é seguida: muitas vezes, tarefas não podem ser consideradas de forma isolada para as atividades seguintes à implementação no fluxo de valor.

Por exemplo, se uma tarefa for "implementar o *back-end* da manutenção de dados pessoais", não é possível verificar o atendimento da DoD (que comumente envolve assegurar que os critérios de aceite da história de usuário como um todo sejam atendidos) para essa tarefa de forma isolada antes de ter também a implementação do *front-end*. Ou seja, a história de usuário, diferentemente da tarefa, representa uma unidade coesa de trabalho no fluxo de valor do Kanban.

Uma dúvida que permanece nesse ponto é sobre como estabelecer de forma adequada os limites WIP para cada atividade. Seguindo a **Lei de Little** da teoria de filas (JAIN, 1990), buscando otimizar a produção, os limites WIP para cada atividade devem ser estabelecidos multiplicando o *lead time* da atividade (tempo médio que uma tarefa fica na atividade) pelo *throughput* (número de cartões produzidos por dia) da atividade com o maior *lead time*.

WIP (ATIVIDADE) = *LEAD TIME* (ATIVIDADE) * *THROUGHPUT*

Vamos assumir que, no nosso exemplo, o *lead time* da especificação seja de 2 dias; o da implementação, de 3 dias, e o da verificação, de

1 dia. Assumindo o *throughput* da tarefa de implementação (a que tem o maior *lead time*) de 2 cartões por dia, teremos:

WIP (especificação) = 2 * 2 = 4

WIP (implementação) = 3 * 2 = 6

WIP (verificação) = 1 * 2 = 2

Não por coincidência, esses números refletem exatamente os limites WIP das atividades do quadro Kanban que vimos no exemplo anterior. Quando os resultados do cálculo do WIP derem números quebrados, eles podem ser arredondados para cima.

3.6 Métodos ágeis escalados — o framework SAFe®

O **SAFe** (*Scaled Agile Framework*) é um framework para a aplicação do ágil em escala amplamente utilizado pelas organizações. Suas práticas estão disponíveis gratuitamente (scaledagileframework.com), formando uma base de conhecimento de princípios, práticas e competências para alcançar a agilidade de negócios em escala.

O SAFe® trabalha com base na orientação ao valor e defende o alinhamento das estruturas organizacionais à cadeia de valor, buscando a otimização constante em tudo que não está alinhado com o valor entregue aos clientes. O termo *agilidade de negócios* significa que toda a organização (e não apenas o time de desenvolvimento) está engajada em entregar soluções de negócio inovadoras, de forma contínua e proativa. Isso permite que a empresa seja mais competitiva, pois seus colaboradores são

alocados às atividades da melhor forma possível, buscando a maximização do valor gerado pelos produtos entregues.

Para isso, é necessário repensar a gestão de portfólio e governança da empresa, de forma a descentralizar a tomada de decisão. Em vez de focadas nos departamentos da empresa, as entregas são organizadas em fluxos de valor. Dentre os princípios do SAFe, podemos destacar: o desenvolvimento incremental, com ciclos rápidos e integrados de aprendizado; a aplicação da cadência e da sincronização dos planejamentos entre domínios e a organização em torno da entrega de valor. A figura a seguir resume os componentes essenciais do SAFe. Os principais deles serão detalhados a seguir.

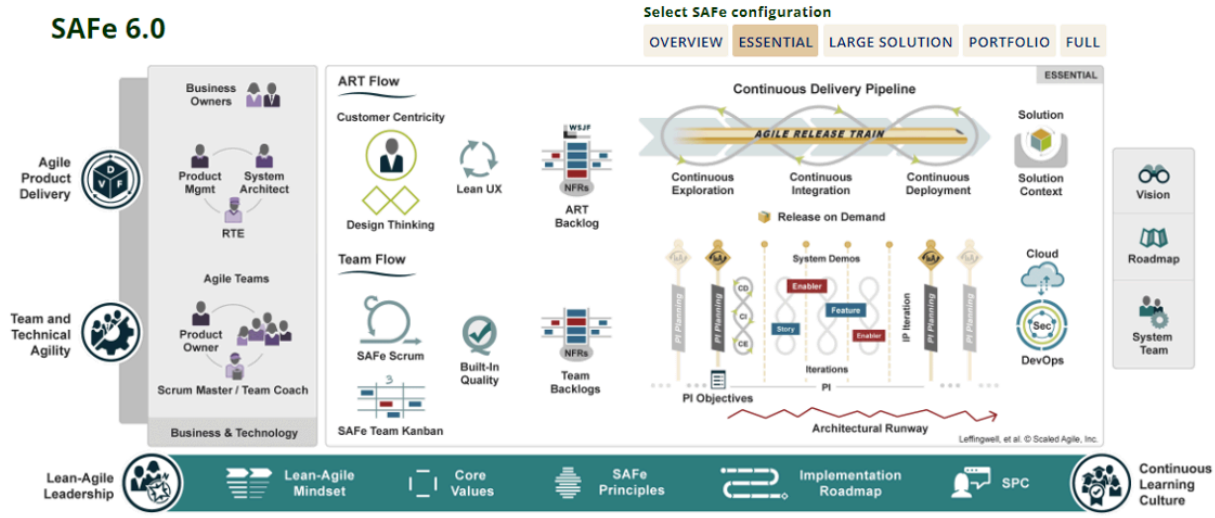


Figura 3.11: Componentes essenciais do SAFe. Fonte: https://www.scaledagileframework.com/.

A principal estrutura de operacionalização do SAFe® é o **trem** (*Agile Release Train* — ART), que consiste em um time de times ágeis que reúne as capacidades necessárias para gerar valor para o negócio (*continuous exploration*) e que trabalha em conjunto em um fluxo de entregas contínuas (*continuous delivery*). O trem desenvolve, integra e opera (quando aplicável) uma ou mais soluções (chamadas de *produtos*) dentro de uma cadeia de valor (*value stream*), que representa uma série de etapas que uma organização segue para implementar soluções que fornecem um fluxo contínuo de valor para

um cliente. Cada *value stream* pode ter um ou mais trens e define um conjunto de KPIs (*Key Performance Indicators*).

Nesse contexto, o **Business Owner (BO)** é o responsável por definir direcionadores de negócio e visão estratégica para direcionar a capacidade de um trem, sendo o responsável pelo patrocínio e engajamento estratégico. Para cada trem, temos:

- Um **Product Manager (PM)**, responsável por manter a visão dos produtos trabalhados no trem, definir e priorizar o *backlog* do trem como um todo e proteger os objetivos do negócio, garantindo o alinhamento destes com o que está sendo desenvolvido e entregue pelos times do trem;
- Um **Release Train Engineer (RTE)**, o líder servidor que atua como *coach* do trem e é responsável por facilitar sues processos e eventos, assegurar a integração entre os times e alinhar a execução à estratégia em conjunto com o PM;
- Um **Solution Architect/Engineer**, responsável por definir e comunicar uma visão técnica e arquitetural compartilhada no trem.

Para cada **time** do trem, temos basicamente a mesma composição que já vimos para o Scrum:

- Um **Product Owner (PO)**, responsável por priorizar as histórias de usuário no time, colaborar com o time nas construções e validar as entregas realizadas, mantendo a integridade conceitual e técnica das funcionalidades ou componentes para o time. É o elo entre o time e o negócio, alinhando as entregas com os objetivos e com a estratégia;
- Um **Scrum Master (SM)**, também chamado de *Agile Master*, responsável por facilitar os rituais do time, gerenciar os impedimentos e o progresso e auxiliar na prática da metodologia ágil;
- Além do PO e do SM, podem compor o time profissionais como desenvolvedores, técnicos e analistas com conhecimento sobre o produto trabalhado no time. O time deve ter autonomia para

definir, construir, testar e entregar incrementos de valor para um determinado produto.

A figura a seguir ilustra as cerimônias que ocorrem no trem e nos times. As principais cerimônias serão detalhadas a seguir.

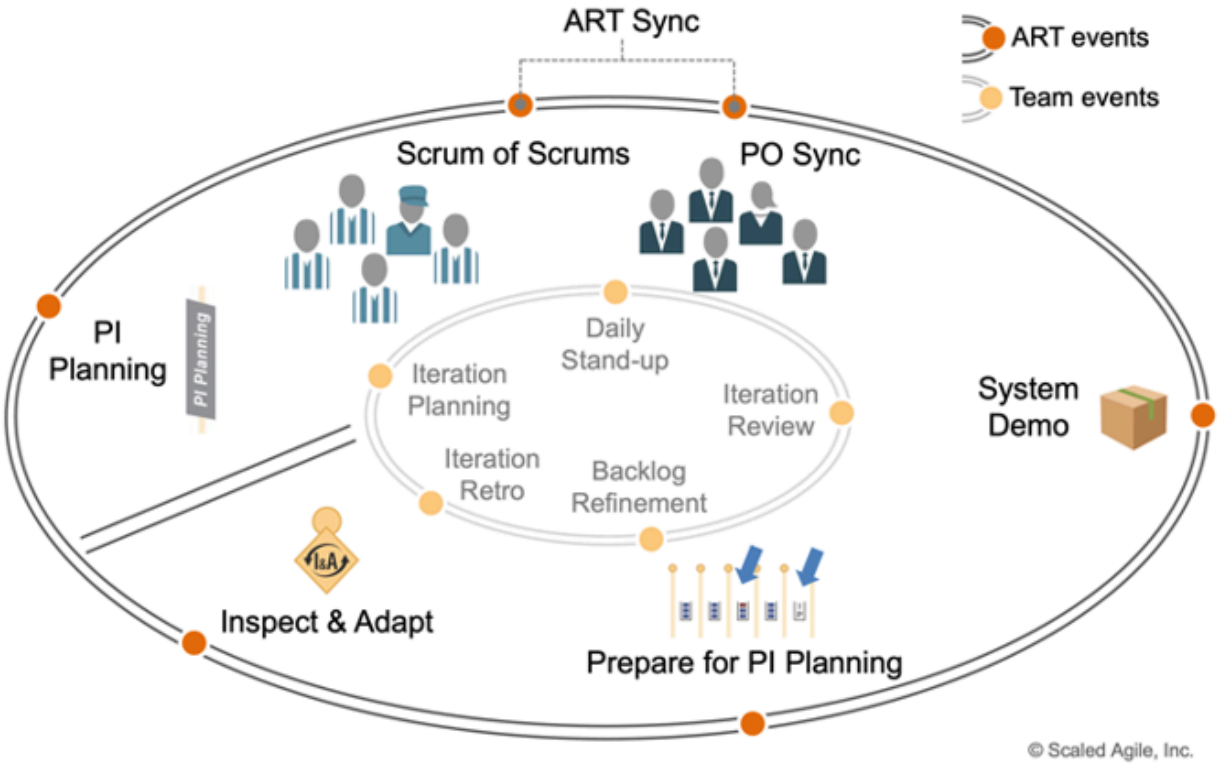


Figura 3.12: Cerimônias do SAFe. Fonte: https://www.scaledagileframework.com/.

Cada ciclo do trem é conhecido por *Program Increment* (PI), um *timebox* de 8 a 12 semanas no qual um trem entrega valor através de softwares funcionais testados. Um PI segue o ciclo PDCA (*Plan, Do, Check, Act*) e é iniciado pela cerimônia *PI Planning* (PIP), um evento de planejamento que geralmente dura 2 dias e tem como participantes todos os envolvidos (BO, PM, RTE, Solution Architect/Engineer e os times).

No PIP, alinha-se a visão do negócio, o *backlog* a ser desenvolvido no PI e as capacidades dos recursos disponíveis, elaborando-se um plano capaz de entregar os objetivos definidos para o ciclo. Em seguida, para cada time ocorrem diversas iterações, sendo comum

que cada PI tenha de 4 a 6 iterações de 2 semanas cada, representando as *sprints* do Scrum. Ao final de cada iteração, pode ocorrer o *System Demo*, um evento que fornece uma visão integrada das novas *features* entregues pelos times na iteração, possibilitando aos *stakeholders* medirem o progresso do PI de forma objetiva, comparando a valoração dos objetivos entregues com a valoração dos objetivos planejados no PIP.

Ao final de cada PI ocorre o evento *Inspect and Adapt* (I&A), quando o estado atual de cada produto é demonstrado e avaliado pelo trem através de uma *System Demo* final. Neste mesmo evento, os times conhecem os resultados de algumas métricas que designam e promovem a "saúde do trem", bem como fazem uma sessão de *problem-solving* para identificar itens de melhoria que podem ser adicionados ao planejamento do próximo ciclo, ao *backlog* do próximo PI.

Tipicamente, o RTE facilita uma reunião semanal de 30 a 60 minutos conhecida como *Scrum of Scrums* (SoS), que ajuda a coordenar as dependências entre os times e prover visibilidade dos progressos e impedimentos. Além do RTE, participam dessa reunião os *Scrum Masters* dos times do trem. De forma similar, o PM facilita semanalmente uma reunião semanal também de 30 a 60 minutos com os POs dos times, conhecida como *PO Sync*, com o objetivo de ter visibilidade do progresso de cada time em direção aos objetivos do PI, discutir problemas e oportunidades e realizar ajustes de escopo, se necessário.

Essa reunião também pode ser usada para realizar a preparação para o próximo PI, o que pode incluir o refinamento e a priorização do *backlog*. Em alguns casos, pode ser interessante unificar a *Scrum of Scrums* e a *PO Sync* em uma única reunião, denominada *ART Sync*. Dentro de cada time, é possível que se utilize o Kanban, o Scrum ou uma combinação de ambos, sendo escolhido o método que se adequar às necessidades do time. No caso de se utilizar Scrum, geralmente são conduzidas internamente cerimônias similares às cerimônias típicas do Scrum, tais como planejamento do

sprint, reunião diária, revisão do *sprint* e retrospectiva do *sprint*, ilustradas pela figura a seguir da documentação oficial.

Tipicamente, quando o método Scrum dividido em *sprints* não é aplicável, os times trabalham com quadros Kanban para gerenciar o *backlog* e o seu fluxo de trabalho em cada nível.

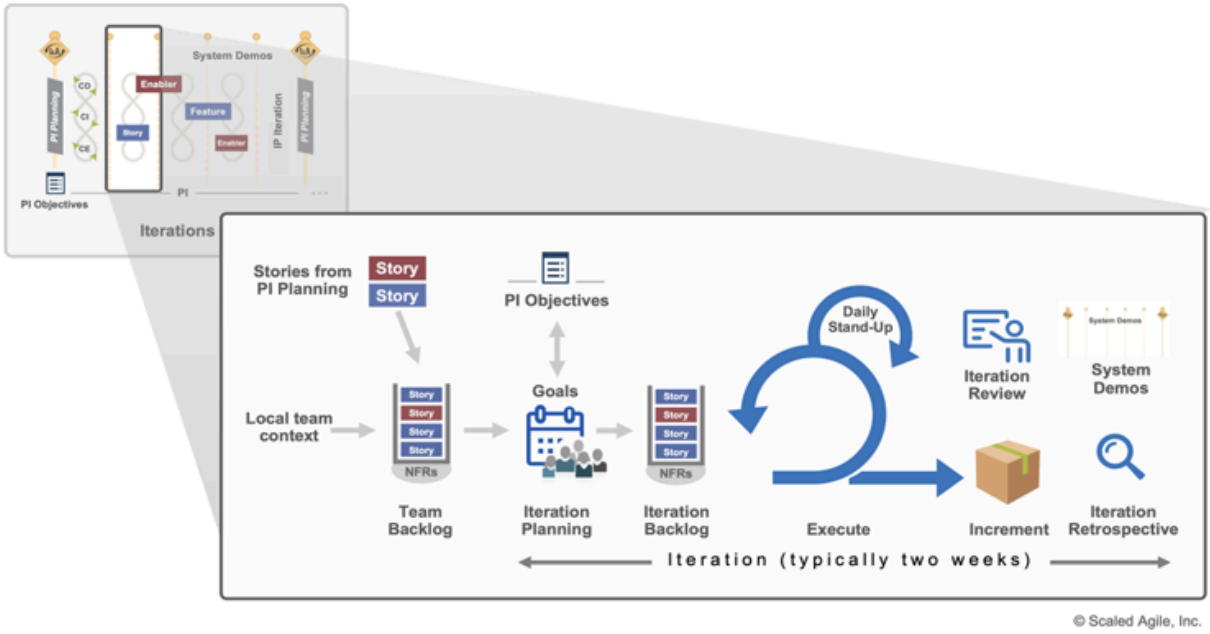


Figura 3.13: SAFe utilizando Scrum. Fonte: https://www.scaledagileframework.com/.

3.7 SAFe® em projetos de Ciência de Dados

Vimos que cada time de um trem do SAFe tem composição basicamente similar à do Scrum, podendo ainda executar cerimônias semelhantes. Assim, para cada time, as recomendações anteriormente fornecidas para a adaptação do Scrum para sistemas inteligentes podem ser essencialmente reaproveitadas.

Em uma perspectiva mais organizacional, vimos que o PM alinha o trabalho dos POs dos diferentes times de um trem (nas *PO Syncs*), enquanto o RTE alinha o trabalho dos *Scrum Masters* (nas *Scrum of*

Scrums). O PM, envolvido na gestão de produtos que envolvem sistemas de software inteligentes, deve alinhar a produção dos produtos à estratégia organizacional, ou seja, ao tipo de produto que gera valor para a organização e que ela gostaria de ter no futuro. Além de ter visão e realizar o planejamento estratégico dos produtos, o PM se beneficia significativamente de entender bem sobre requisitos de sistemas de software inteligentes, até para assegurar que as características dos diferentes produtos sendo produzidos pelos times tenham convergência para o alcance dos objetivos estabelecidos para os produtos pela organização.

O entendimento das possíveis características para estes produtos pode ser obtido através do estudo da técnica PerSpecML (VILLAMIZAR *et al.*, 2023). Entretanto, como o PM não realiza a especificação, é fortemente recomendado conhecer ao menos um catálogo de características (como o disponível em Villamizar *et al.*, 2022a).

Apenas para exemplificar, imagine um PM, responsável por definir a visão para produtos de software inteligentes, que não entenda os conceitos relacionados a esse tipo de produto, como medidas típicas correlacionadas com o sucesso futuro (*leading indicators*), contundência (*forcefulness* — usada para medir até que ponto a inteligência artificial construída será impositiva na experiência do usuário), formas de fornecimento dos modelos para os produtos, explicabilidade ou aspectos éticos relacionados aos dados. Note que citamos apenas cinco de mais de 50 características que podem precisar ser levadas em consideração no contexto de sistemas de software inteligentes — mas já é possível perceber que, sem esse conhecimento, o trabalho do PM ficará significativamente prejudicado.

Ainda na perspectiva organizacional, a pessoa *Solution Architect/Engineer* é responsável por alinhar aspectos arquiteturais entre os times, assegurando não só uma *stack* de tecnologias alinhada aos objetivos, mas também o tipo de arquitetura a ser seguido na construção da solução. É natural que no contexto de

sistemas de software inteligentes envolvendo *Machine Learning* ela se aprofunde a respeito de alternativas para arquiteturas para esse tipo de sistema. Ela deverá saber, por exemplo:

- As diferentes formas de implantação e fornecimento de modelos, bem como as vantagens e desvantagens de uma implantação em nuvem;
- Como elaborar arquiteturas que considerem aspectos de MLOps para situações em que modelos necessitem de retreinamento automático;
- Conhecer alternativas de *frameworks* e ferramentas (*dashboards*) para visualização da informação;
- Entender de previsões off-line, em tempo real, síncronas e assíncronas;
- Conhecer alternativas para otimizar a predição dos modelos.

Esses assuntos específicos de Ciência de Dados não são cobertos nos livros tradicionais de Engenharia ou Arquitetura de Software e tentamos ajudar a fechar esse *gap*, compilando conhecimento sobre arquiteturas para esse tipo de sistema obtido tanto do estudo da área quanto da vivência, entregando dezenas de soluções de sistemas inteligentes ao longo dos últimos anos.

Temos aplicado o SAFe® em diversos projetos, em particular nos projetos da iniciativa ExACTa PUC-Rio em parceria com a Petrobras envolvendo inteligência artificial na operação. Os excelentes resultados obtidos indicam que essa abordagem é adequada para escalar agilidade em uma grande corporação, ajudando a sincronizar as atividades de diversos trens com diversas equipes a fim de maximizar o valor gerado pelos produtos entregues.

3.8 BizDev, DevOps e experimentação contínua

Os avanços na área de *Machine Learning* e na capacidade de processamento e a disponibilidade cada vez maior de dados têm tornado possível, em muitos casos, substituir funções cognitivas humanas e automatizar decisões de forma escalável com base em aprendizado automático obtido a partir de dados. Não é à toa que sistemas de software inteligentes, que incorporam componentes de *Machine Learning*, têm se mostrado cada vez mais viáveis e têm sido considerados com frequência em contextos de inovação e transformação digital das empresas.

De forma resumida, a transformação digital pode ser vista como um processo em que empresas investigam o uso de tecnologias digitais para inovar sua forma de operação, buscando resolver problemas de negócio e alcançar objetivos estratégicos (KALINOWSKI *et al.*, 2020). Mas o que é inovar? Uma definição interessante é do MIT e que nos foi apresentada na visita do Dr. Phil Budden à iniciativa ExACTa PUC-Rio: "*Inovação é o processo de levar uma ideia do seu princípio (_inception*) até o impacto_" (BUDDEN; MURRAY, 2023).

O alinhamento do desenvolvimento de soluções de software com a estratégia de negócio (onde nascem as ideias) e com a operação (onde elas obtêm impacto) tem sido o tema da Engenharia de Software Contínua (FITZGERALD; STOL, 2017). Na Engenharia de Software Contínua, conforme pode ser visto na figura a seguir, práticas de **BizDev** buscam o alinhamento entre a estratégia de negócio e o desenvolvimento, enquanto as práticas de **DevOps** representam o alinhamento entre o desenvolvimento e a operação. A **experimentação contínua** (também conhecida no mercado como *teste A/B*), por sua vez, trata o ciclo de retroalimentação da operação para a estratégia de negócio.

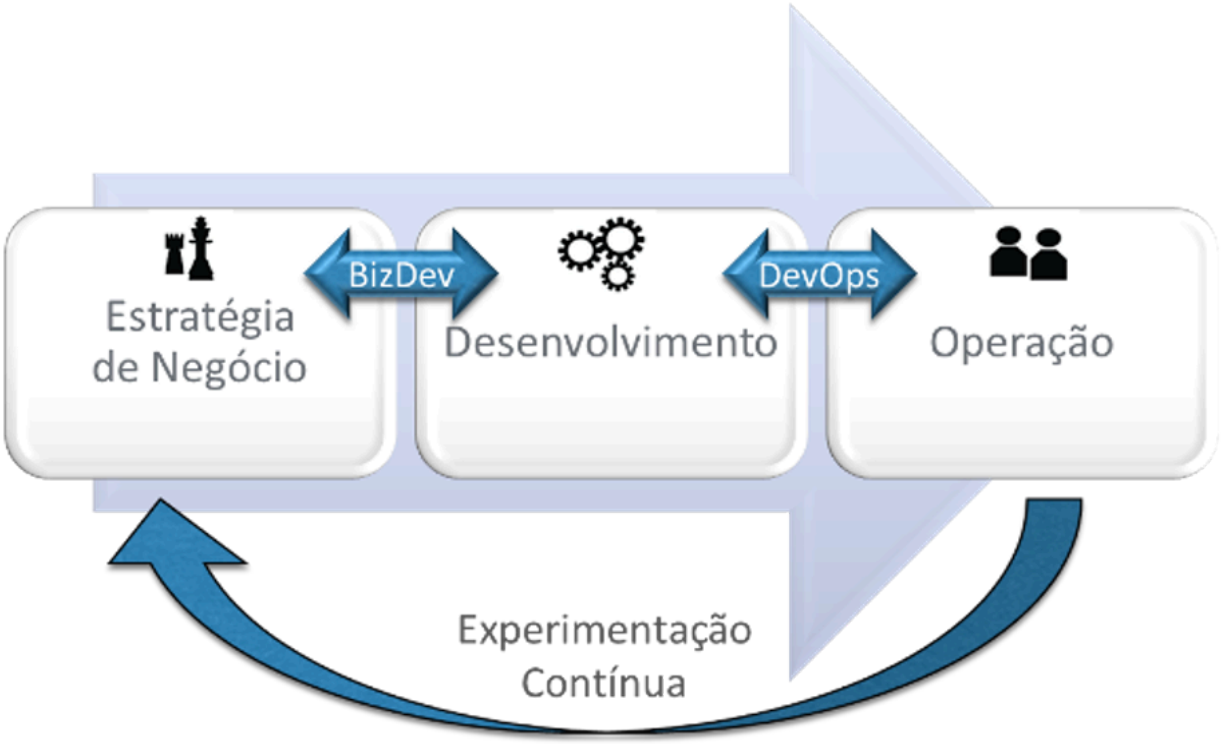


Figura 3.14: Experimentação contínua.

Para exemplificar, em contextos de transformação digital, normalmente se busca identificar um **produto mínimo viável** (ou **MVP – *Minimum Viable Product***) de software que permita testar hipóteses de negócio alinhadas com a estratégia de negócio da empresa. Este produto precisa ser desenvolvido refletindo esse alinhamento com a estratégia e ser posto em operação junto aos seus usuários finais para que as hipóteses de negócio possam ser de fato testadas. O teste das hipóteses de negócio com o software em operação é a experimentação contínua, fechando um ciclo de retroalimentação. É essa avaliação que permitirá planejar incrementos (ou alternativas) para o produto para buscar melhorar cada vez mais os resultados de negócio. A busca pela melhoria resulta em novas hipóteses de negócio, a serem testadas no próximo ciclo de experimentação contínua.

3.9 MLOps e DataOps

Em sistemas de software inteligentes que incorporam componentes de *Machine Learning*, não basta somente alinhar o desenvolvimento do software com a operação (DevOps): é preciso também manter os modelos de *Machine Learning* atualizados e representativos para a operação a todo instante. Há diversas situações em que fenômenos que afetam o comportamento dos dados fazem com que as predições dos modelos deixem de fazer sentido.

Durante a pandemia, por exemplo, as pessoas mudaram seus padrões de consumo. Em relação ao vestuário, em geral, passaram a comprar menos roupas sociais e mais roupas casuais. Isso indica uma mudança de comportamento que um modelo treinado antes da pandemia não teria como prever. Existe, então, a necessidade de identificar essas mudanças o quanto antes e de retreinar e reimplantar uma nova versão do modelo, mesmo que outras partes do sistema não sejam alteradas.

MLOps é o alinhamento da construção dos modelos de *Machine Learning* com o desenvolvimento de software e a operação. Assim, contempla as melhores práticas para colaboração entre cientistas de dados, desenvolvedores e profissionais de operação. A figura a seguir ilustra a necessidade de se alinhar os modelos com o desenvolvimento e com a operação.

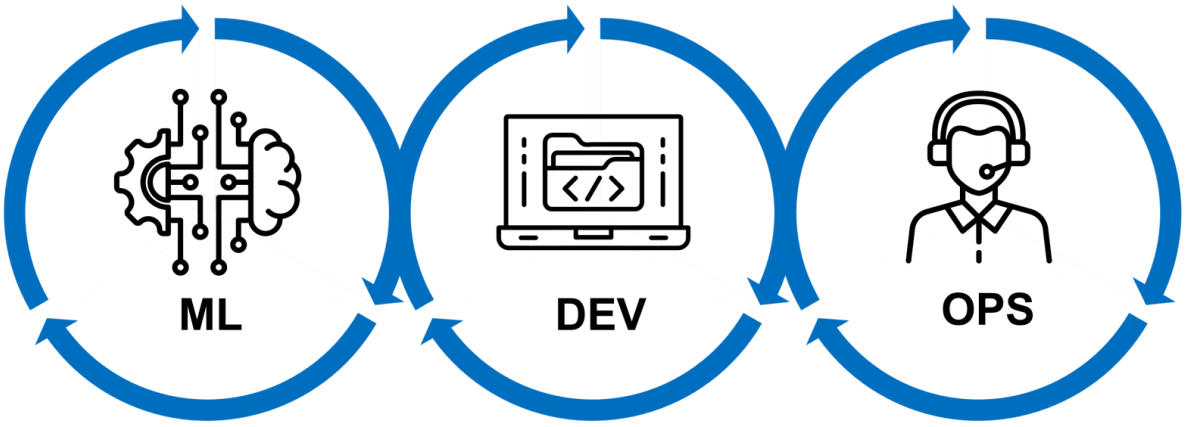


Figura 3.15: MLOps.

Muitas vezes não se pode simplesmente implantar um modelo de *Machine Learning* treinado off-line como um serviço de previsão definitivo. Em diversas situações, será necessário um *pipeline* de várias etapas para retreinar e implantar automaticamente um modelo de forma contínua. Esse *pipeline* adiciona complexidade porque precisa automatizar etapas que os cientistas de dados normalmente executam manualmente antes da implantação para treinar e validar novos modelos.

As fases típicas de um *pipeline* de MLOps contemplam: coleta e análise de dados; preparação dos dados; treinamento do modelo; validação do modelo; colocar o modelo em operação; monitorar o modelo e retreiná-lo sempre que necessário. Normalmente o que dispara a necessidade de uma atualização do modelo é a identificação de um fenômeno conhecido como *concept drift* durante a monitoração do modelo. Dizemos que ocorreu um *concept drift* quando mudanças no comportamento dos dados são identificadas. Assim, é comum que na etapa de monitoração do modelo haja mecanismos de identificação de *concept drift*, a fim de disparar o *pipeline* de MLOps para, por exemplo, iniciar o retreinamento do modelo com novos dados.

Ainda em relação aos dados, **DataOps** envolve zelar pela qualidade dos dados em todos os aspectos e fornecê-los de forma confiável para uso, seja na operação ou para atividades como a construção de

modelos. Assim, conforme ilustrado na figura a seguir, DataOps visa unir as áreas de Engenharia de Dados, Integração de Dados, Qualidade de Dados e Segurança e Privacidade de Dados para o sucesso dos projetos.

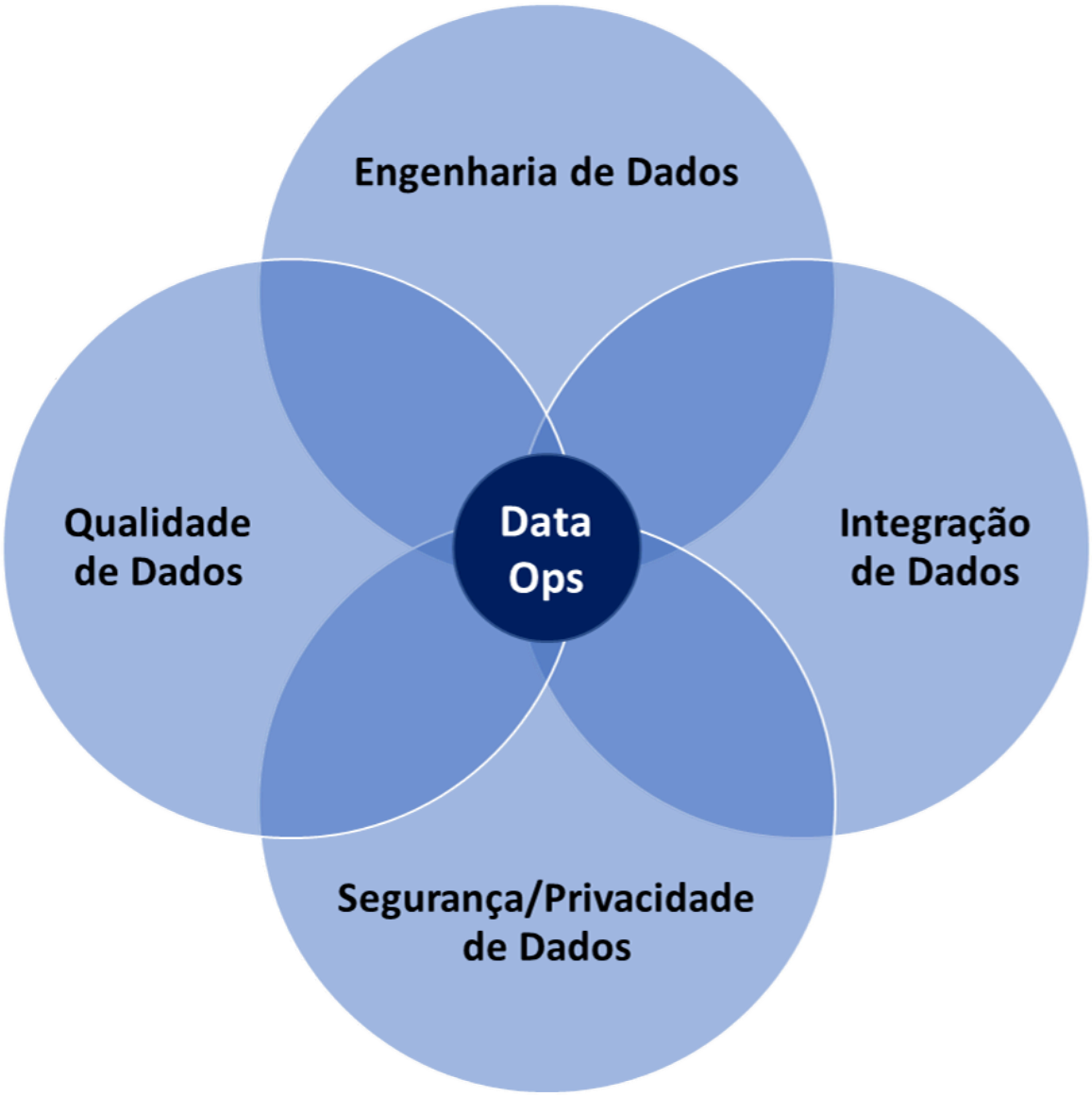


Figura 3.16: DataOps.

3.10 Abordagem Lean R&D

A *abordagem Lean R&D* foi criada e é utilizada na iniciativa ExACTa PUC-Rio para o desenvolvimento de sistemas de software inteligentes. **Lean** remete a uma abordagem ágil de desenvolvimento enxuto, sem desperdício. Já o **R&D** (de *Research and Development*, ou **P&D**, de *Pesquisa e Desenvolvimento*) remete a inovação e a lidar com incertezas inerentes a projetos de pesquisa e desenvolvimento. Lean R&D foi pensada para ser uma abordagem de cocriação de P&D ágil inspirada nos preceitos da Engenharia de Software Contínua e do *Lean*, centrada na experimentação contínua para a avaliação do atendimento de hipóteses de negócio em ciclos curtos.

A abordagem ajudou a quebrar na prática o paradigma de que projetos de P&D não se adequam a métodos ágeis com entregas rápidas e frequentes. Ela nos auxiliou a realizar dezenas de entregas de P&D em intervalos de poucos meses. Em 2021, com o uso da abordagem, a iniciativa ExACTa foi vencedora do prêmio Reconhecimento à Inovação do CENPES 2021, o centro de pesquisa da Petrobras, na categoria que seleciona empresas parceiras que geraram entregas rápidas e de valor.

Embora a abordagem tenha sido pensada para projetos de P&D, na prática temos aplicado ela essencialmente para a construções de soluções de Ciência de Dados, envolvendo a construção de sistemas de software inteligentes. De fato, projetos de Ciência de Dados, assim como projetos de P&D, também tendem a ter incertezas ao longo da sua construção e estão comumente relacionados com contextos de inovação, o que normalmente envolvem aplicar tecnologias em contextos nos quais até então ainda não foram aplicadas, por exemplo, substituindo atividades cognitivas humanas por soluções de *Machine Learning*. A seguir detalharemos os papéis e uma visão geral da abordagem.

Papéis do Lean R&D

A abordagem conta com os seguintes papéis:

- **Comitê Gestor**: Avalia os projetos nos pontos de verificação descritos. Essa avaliação visa: (i) permitir a "falha rápida" de ideias que não entregariam o valor de negócio esperado; e (ii) assegurar que a abordagem está sendo usada para enfrentar desafios de inovação e transformação digital com alto potencial de impacto;
- ***Scrum Master***: Facilita as *Lean Inceptions* e gerencia as equipes de pesquisa e desenvolvimento ágeis, garantindo que a abordagem geral de P&D ágil seja seguida de forma adequada;
- ***Product Owner* (PO) e representantes do cliente**: Assim como no Scrum tradicional, os POs são responsáveis por maximizar o valor comercial do produto resultante do trabalho da equipe de desenvolvimento;
- **Desenvolvedores**: A equipe de desenvolvimento;
- **Cientistas de dados**: Apoiam a equipe de desenvolvimento em uma avaliação inicial de viabilidade técnica e em tarefas relacionadas com Ciência de Dados durante o desenvolvimento (por exemplo, investigando técnicas de *Machine Learning* a serem usadas, elaborando modelos de previsão);
- **Equipe de design de UX/UI**: Responsável por projetar *mock-ups* de interação com o usuário e protótipos de alta fidelidade para subsidiar o desenvolvimento do *front-end*;
- **Analista de DevOps e infraestrutura**: Responsável por fornecer a infraestrutura DevOps para as equipes de desenvolvimento.

Visão geral do Lean R&D

Inspirada no Lean, a abordagem Lean R&D busca maximizar o valor de negócio entregue minimizando o desperdício. Além disso, ela aproveita o conceito do *fail fast* da abordagem *Lean Startup* (RIES, 2011), possibilitando lidar com oportunidades e riscos. O *fail fast* preconiza que assim que se descubra que uma ideia não vingará, ela deve ser interrompida, possibilitando pensar em novas ideias. Cabe ressaltar que temos utilizado a abordagem em projetos de

transformação digital e inovação, em que o primeiro passo é a **ideação** de um novo produto a ser construído.

Uma outra característica é conter adaptações e suportes para permitir lidar com incertezas e problemas complexos, como tipicamente ocorre em projetos de P&D e na construção de sistemas inteligentes. A figura a seguir apresenta uma visão geral da abordagem *Lean R&D*. É possível ver na figura como os conceitos de conexão e continuidade da Engenharia de Software Contínua (BizDev, DevOps e experimentação contínua) se fazem presentes. Na sequência, apresentaremos cada uma das suas fases.

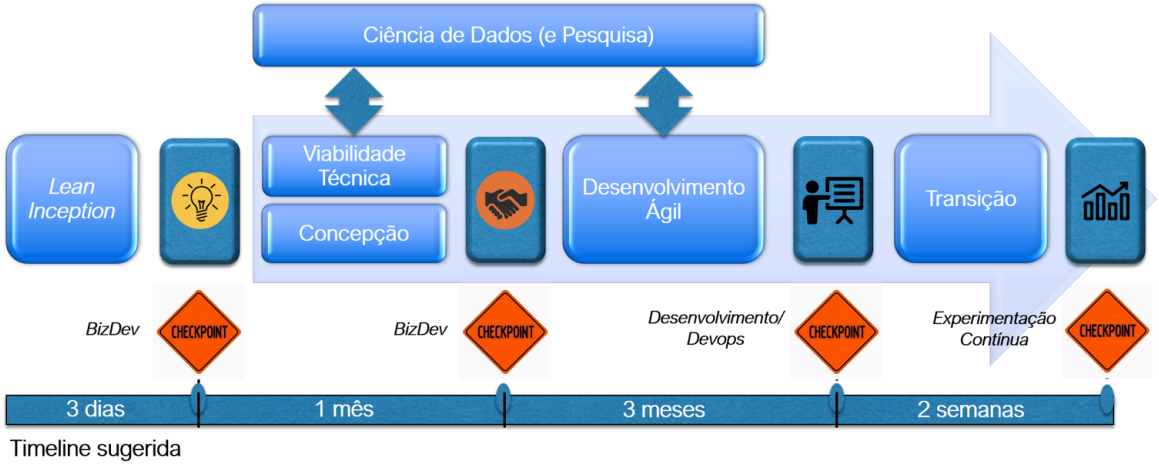


Figura 3.17: Visão geral da abordagem Lean R&D (KALINOWSKI *et al.*, 2020).

Os pontos de verificação (*checkpoints*) desempenham um papel chave na abordagem, permitindo aplicar o conceito do *fail fast*. Considerando isso, vamos descrever a abordagem nos ancorando nos pontos de verificação.

A abordagem começa com uma ***Lean Inception*** (CAROLI, 2018) para permitir que as partes interessadas definam em conjunto a visão de um MVP que pode ser usado para testar hipóteses de negócios (detalharemos mais a *Lean Inception* no próximo capítulo deste livro). Cabe ressaltar que é possível adaptar a abordagem para o uso de outras dinâmicas de ideação, desde que o resultado ajude

a entender o produto a ser construído e inclua hipóteses de negócio a serem testadas.

Algo que pode gerar estranheza na figura é a *Lean Inception* sendo conduzida em três dias em vez de em cinco dias, como recomendado pelo criador do método. Isso é reflexo de uma adaptação exigida por parte do nosso principal parceiro industrial quando estávamos definindo a abordagem. Sabemos que, caso haja disponibilidade dos interessados do lado cliente, os dois dias adicionais podem de fato representar um excelente investimento.

No **primeiro ponto de verificação**, o comitê gestor deve aprovar a visão definida para o MVP. Se for rejeitada, uma nova *Lean Inception* deve ser conduzida, potencialmente focando em um problema diferente ou em uma forma completamente diferente de resolver o problema. Se for aprovada, duas fases começam em paralelo: **viabilidade técnica** e **concepção**.

Durante a **viabilidade técnica**, a equipe de desenvolvimento, auxiliada pela equipe de Ciência de Dados e o analista de infraestrutura, começa a investigar a viabilidade técnica de implementação das *features* identificadas durante a *Lean Inception*. Seguindo a estratégia *tracer bullet*, esta fase normalmente serve como uma prova de que a arquitetura é compatível e viável e de que há uma maneira de resolver o problema com eficácia razoável, além de fornecer um esqueleto funcional demonstrável (MVP0) contendo algumas implementações iniciais.

A **concepção** envolve o PO detalhando as *features* do MVP identificadas durante a *Lean Inception*, aplicando dinâmicas de *Product Backlog Building* com os representantes do cliente, seguida por outras técnicas típicas de elicitação de requisitos (por exemplo, entrevistas), a fim de especificar histórias de usuário. A equipe de UX/UI participa da criação de protótipos de baixa fidelidade, para validação de requisitos, e protótipos de UI de alta fidelidade, para testes de usabilidade. As histórias de usuário devem ser complementadas com cenários de *Behavior-Driven Development*

(BDD), que devem ser usados como critério de aceitação. Ao final da concepção, ocorre a revisão dos requisitos ágeis (histórias de usuários, cenários BDD e *mock-ups*) e os testes de usabilidade nos protótipos de alta fidelidade, ambos junto ao cliente.

No **segundo ponto de verificação**, o comitê gestor deve decidir se o MVP deve ser realmente desenvolvido, analisando a especificação de requisitos concebida, juntamente com os resultados da avaliação de viabilidade técnica, da revisão de requisitos e dos testes de usabilidade. Caso decidam pelo desenvolvimento, inicia-se a fase de **desenvolvimento ágil**.

A fase de **desenvolvimento ágil** envolve os desenvolvedores, com o apoio da equipe de Ciência de Dados, implementando o MVP. Essa fase segue o desenvolvimento baseado em Scrum padrão com planejamento de *sprint*, reuniões diárias, revisões e retrospectivas de *sprint*. Para fins de controle da qualidade, recomendamos o uso de revisões de código modernas, que permitem identificar defeitos, melhorar soluções e compartilhar conhecimento e propriedade do código. Recomendamos também a criação de um painel de gestão ágil que permita transparência sobre o progresso geral da equipe.

No **terceiro ponto de verificação**, o comitê gestor deve decidir sobre a implantação do MVP em produção, após apresentação detalhada do PO. Decidindo pela implantação, inicia-se a fase de **transição**.

A fase de **transição** envolve a equipe de desenvolvimento e infraestrutura, preparando o MVP para o teste beta em seu ambiente final e avaliando as hipóteses de negócios. A equipe de pesquisa deve projetar o plano de experimento, que deve delinear como instrumentar o produto para permitir a coleta das medidas necessárias para testar as hipóteses de negócios e, eventualmente, construir outros instrumentos de avaliação — por exemplo, questionários ou uma pesquisa de NPS (*Net Promoter Score*) embutida diretamente no produto para medir a satisfação do usuário.

No **último ponto de verificação**, o comitê gestor analisa os resultados da experimentação contínua — por exemplo, se as hipóteses de negócios foram alcançadas — e decide sobre investir em outro ciclo para melhorar o MVP, produzindo novos incrementos. Caso decidam por um novo ciclo, a *Lean Inception* pode ser conduzida de forma mais focada ou até substituída por uma dinâmica mais enxuta, focada em delinear possíveis incrementos para o MVP.

Cabe ressaltar que *Lean R&D* prescreve o emprego de diversas boas práticas da Engenharia de Software para maximizar as chances de sucesso dos projetos, tais como:

- Gestão Ágil;
- Práticas de DevOps e MLOps;
- Integração apropriada do time de entrega envolvendo desenvolvedores, cientistas de dados, designers de UX/UI e analistas de DevOps;
- Avaliação inicial da viabilidade técnica e arquitetural;
- Especificação antecipada de requisitos ágeis com critérios de aceitação;
- Design antecipado da experiência do usuário e das interfaces;
- Revisão de requisitos e testes de usabilidade;
- Boas práticas de projeto e construção;
- Análise estática e testes automatizados;
- Revisões de código modernas;
- Gerência de configuração;
- Integração e implantação contínua, entre outras.

Mais detalhes podem ser encontrados no artigo que descreve a abordagem (KALINOWSKI *et al.*, 2020).

Embora Lean R&D tenha sido recentemente publicada (e, até então, somente em meios científicos), já temos conhecimento de empresas e colaborações de P&D que têm começado a utilizar Lean R&D com sucesso. Além de sermos usuários assíduos da nossa própria abordagem, que tem nos permitido realizar P&D ágil em cocriação

junto aos nossos parceiros da indústria, temos apresentado e ajudado a implantar a abordagem em empresas envolvidas com projetos de pesquisa e Ciência de Dados dentro e de fora do país.

Neste capítulo, vimos como métodos ágeis podem ser aplicadas no contexto da Engenharia de Sistemas de software inteligentes. A escolha da abordagem ou do método a ser seguido dependerá do contexto e da cultura organizacional. Por exemplo, se a organização é pequena e se adapta bem a trabalhar com o conceito de *sprints*, então o Scrum pode ser seguido. Para contextos em que os *sprints* podem não se mostrar adequados, uma alternativa é seguir o Lean com Kanban.

Para grandes organizações, pode ser necessário empregar métodos ágeis escalados como o SAFe® para alinhar o que está sendo feito nos diferentes projetos. Caso você tenha interesse em se aprofundar em métodos ágeis de maneira geral (sem o nosso foco específico), sugerimos o livro *Métodos ágeis para desenvolvimento de software*, de Prikladnicki *et al.* (2014).

Além disso, vimos que, no contexto de projetos de inovação, conceitos como BizDev, DevOps e experimentação contínua se mostram relevantes para permitir testar hipóteses de negócio. Vimos ainda que sistemas inteligentes comumente possuem preocupações relacionadas com estratégias de MLOps e DataOps. Uma abordagem que pode ser utilizada para projetos de inovação envolvendo sistemas inteligentes é Lean R&D, que se mostra adequada para lidar com incertezas como as encontradas nesse tipo de projeto.

Capítulo 4
Especificação de sistemas de software inteligentes

Este capítulo apresentará os conceitos básicos de Engenharia de Requisitos e como esses conceitos são aplicados em contextos ágeis, que são os predominantes no mercado. Também falaremos de limitações das abordagens atuais, como o ML Canvas, para a especificação de sistemas de software inteligentes.

Apresentaremos ainda a abordagem de especificação de *Machine Learning* baseada em perspectiva, PerSpecML (*Perspective-based Specification of Machine Learning-Enabled Systems*), que foi criada pelos autores deste livro e que cobre uma visão bem mais ampla do que o ML Canvas, podendo ser considerada o estado da arte para a especificação de sistemas baseados em *Machine Learning*.

4.1 Conceitos básicos de Engenharia de Requisitos

> "*A parte mais difícil da construção de um sistema de software é decidir precisamente o que deve ser construído. Nenhuma outra parte do trabalho conceitual é tão difícil quanto estabelecer detalhadamente os requisitos técnicos, incluindo todas as interfaces com pessoas, máquinas e outros sistemas de software*."
>
> — Frederick P. Brooks Jr.

A frase clássica acima, amplamente conhecida na área de Engenharia de Software, foi publicada em uma edição da revista *IEEE Computer* de 1987, mas seu conteúdo é consenso até hoje. De fato, problemas de requisitos têm se mostrado como determinantes para o fracasso de projetos de software (MENDEZ *et al.*, 2017). Quando se trata de requisitos de sistemas de software inteligentes, a dificuldade reportada ainda aumenta. Ao longo deste capítulo vamos abordar a especificação de sistemas de software inteligentes em detalhes e tentar simplificar o processo. Iniciaremos com algumas definições-chave da área de requisitos.

De acordo com o *International Requirements Engineering Board* (IREB), que é o principal órgão de certificação profissional na área de requisitos, o termo **requisito** denota três conceitos (GLINZ *et al.*, 2020):

- Uma necessidade percebida por um interessado;
- Uma capacidade ou propriedade que um sistema deve possuir;
- Uma representação documentada de uma necessidade, capacidade ou propriedade.

Ou seja, não basta especificar o que o sistema fará, é preciso também descrever capacidades ou propriedades relacionadas a outras características do software, como compatibilidade, confiabilidade, desempenho, manutenibilidade, portabilidade, segurança e usabilidade.

É importante ressaltar que há diferentes visões para requisitos. Os **requisitos do usuário** expressam resultados desejados para superar problemas no mundo real. Normalmente eles são especificados de forma mais abstrata, focados no problema e sem prover muitos detalhes do que exatamente será construído. Assim, eles não costumam fornecer detalhes sobre telas e regras de negócio mais específicas.

Os **requisitos do sistema**, por sua vez, focam na especificação das soluções a serem desenvolvidas utilizando sistemas novos e existentes. Eles devem fornecer detalhes sobre *o que* será construído, incluindo a especificação das interfaces e da interação com pessoas, máquinas e outros sistemas de software. Não devem, entretanto, entrar em detalhes sobre *como* o sistema será construído, já que isso será especificado por equipes técnicas durante a arquitetura e o projeto do sistema. Para que um projeto seja considerado bem-sucedido, os requisitos do sistema devem satisfazer os requisitos do usuário, resolvendo o problema que ele enfrenta. Essa relação está representada na figura a seguir.

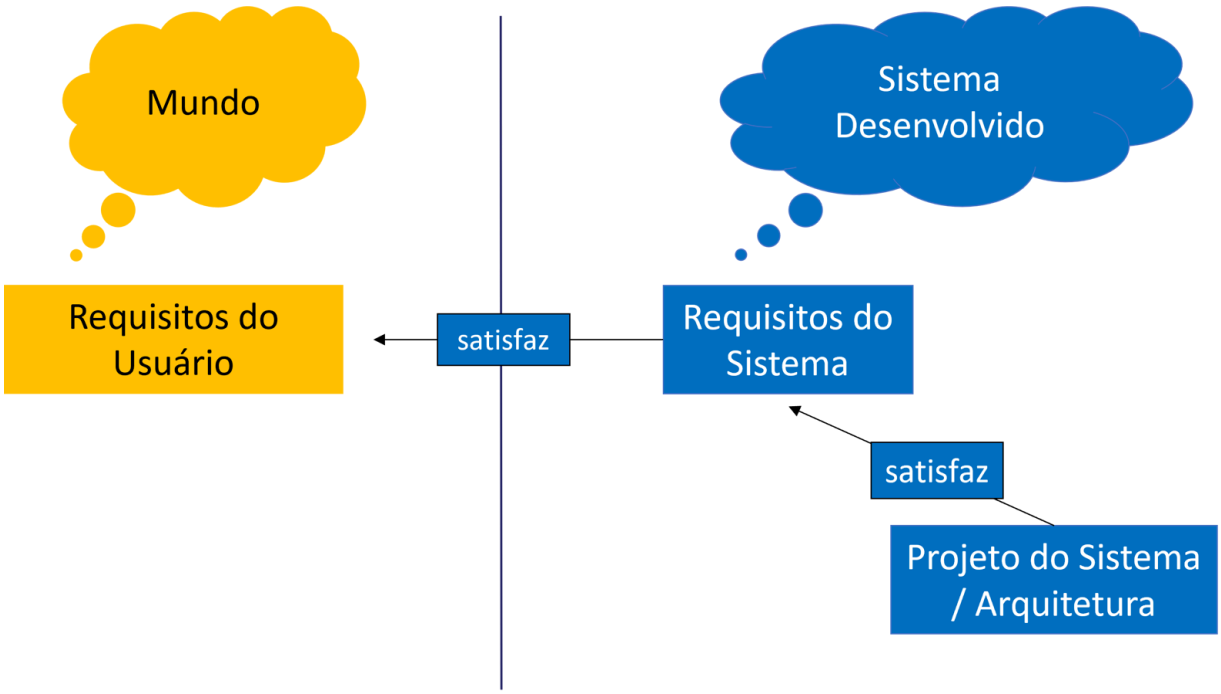


Figura 4.1: Requisitos do usuário x requisitos do sistema.

Em abordagens mais tradicionais, os requisitos do usuário são comumente especificados em documentos conhecidos como *documento de visão* ou *documento de requisitos do usuário*. Já os requisitos do sistema são comumente especificados em documentos conhecidos como *documento de requisitos do sistema*, ou *do software*. Veremos mais adiante como isso é tratado em abordagens ágeis, que são as hoje mais utilizadas pelo mercado, mas, antes disso, precisamos ainda entender o conceito de Engenharia de Requisitos.

A **Engenharia de Requisitos** pode ser definida como " *a abordagem sistemática e disciplinada para a* ***especificação e gerenciamento*** *de requisitos com o objetivo de compreender os desejos e necessidades das partes interessadas e minimizar o risco de entregar um sistema que não atenda a esses desejos e necessidades*" (GLINZ *et al.*, 2020). Ou seja, é a disciplina da Engenharia de Software que compreende a produção e o gerenciamento dos requisitos.

As atividades típicas relacionadas com a produção e a gerência dos requisitos estão ilustradas na figura a seguir. Tipicamente a produção envolve o levantamento, a especificação e a verificação e validação de requisitos. Já a gerência de requisitos envolve a gerência de configuração (versionamento), controle de mudanças, rastreabilidade (manter informações sobre quem pediu determinado requisito e quais partes do sistema ele afeta), e a gerência da qualidade dos requisitos.

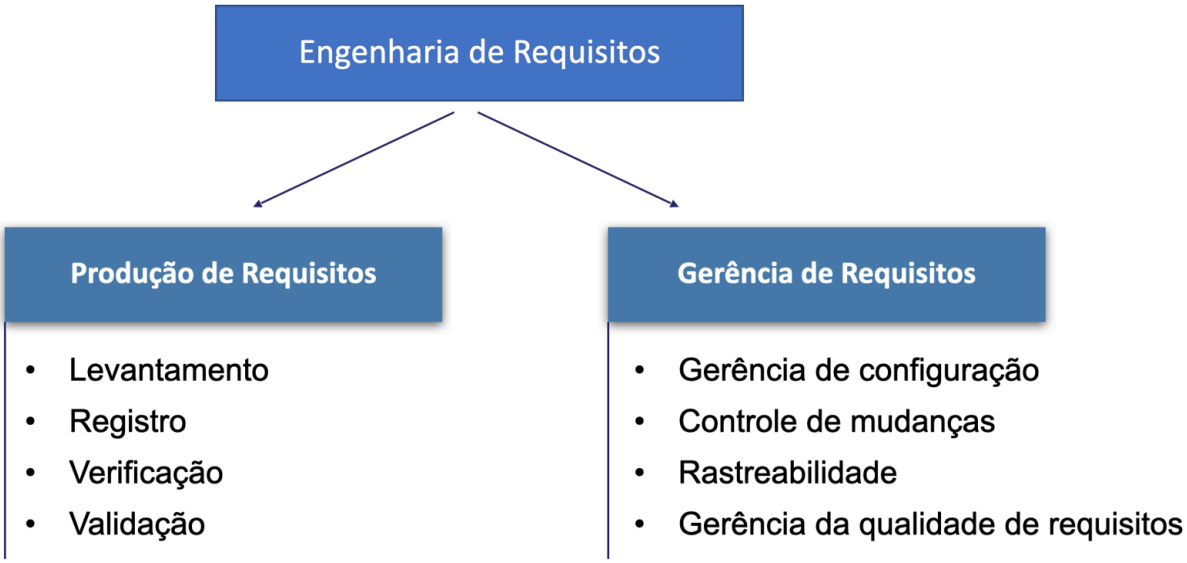


Figura 4.2: Engenharia de Requisitos.

4.2 Tipos de requisitos

Requisitos podem ser divididos nos seguintes três tipos: *requisitos funcionais*, *requisitos não funcionais* e *regras de negócio* (SOMMERVILLE, 2019). Cabe ressaltar que em algumas fontes, os requisitos não funcionais são ainda subdivididos em requisitos de qualidade e restrições (GLINZ *et al.*, 2020). Vamos entender cada um desses tipos de requisitos a seguir.

Os **requisitos funcionais** descrevem as *funcionalidades* que o software deve ser capaz de realizar. Ou seja, são requisitos diretamente ligados à funcionalidade do software. É comum que sejam inicialmente descritos de forma abstrata (como requisitos funcionais do usuário) e depois detalhados (como requisitos funcionais do sistema).

Regra de formação (boa prática):

(ID) O software deve permitir que o (responsável pela ação) + (ação).

Exemplos:

(RF1) O software deve permitir que o gerente financeiro realize a predição do faturamento mensal da loja.

(RF2) O software deve permitir que o gerente do banco consulte a classificação automática de um cliente com base no seu histórico bancário.

Os **requisitos não funcionais**, em vez de informar o que o sistema fará, descrevem **características de qualidade** ou **restrições**. De acordo com o modelo de qualidade de produtos de software ISO 25010, as *características de qualidade* que podem precisar ser especificadas, além dos aspectos funcionais em si capturados pelos requisitos funcionais, são: compatibilidade, confiabilidade, desempenho, manutenibilidade, portabilidade, segurança e usabilidade. As *restrições*, por sua vez, podem ser relacionadas a escolhas de tecnologia da organização — por exemplo, restringir as possibilidades ao uso de um sistema de gerenciamento de banco de dados específico.

EXEMPLO:

(RNF1) A predição realizada pelo sistema deve ser obtida dentro de três segundos. (requisito não funcional de desempenho).

(RNF2) Deve ser possível executar o sistema nas plataformas mobile Android e iOS (requisito não funcional de portabilidade).

É importante ressaltar que os requisitos não funcionais podem ser mais críticos que os requisitos funcionais e que devem ser descritos de forma mensurável e verificável. **Em muitas situações, o sistema que não satisfaz requisitos não funcionais se torna um sistema inútil**. Basta pensar em um sistema de tempo real que não atende às restrições de desempenho necessárias para o seu uso, ou em um sistema voltado para usuários com deficiência sem tratar aspectos da característica de usabilidade referentes à acessibilidade.

Por fim, os **requisitos de domínio** descrevem **regras de negócio que devem ser atendidas pelo sistema**. Normalmente essas regras ajudam a detalhar os requisitos funcionais com aspectos que precisarão ser considerados ou obedecidos em computações específicas. Se os requisitos de domínio não forem satisfeitos, o sistema pode tornar-se não prático.

EXEMPLO:

(RN1) Não podem ser concedidos empréstimos para pessoas físicas que não tenham comprovante de rendimentos.

(RN2) Um cliente de um empréstimo é considerado inadimplente caso o atraso no pagamento de alguma de suas parcelas tenha ultrapassado 30 dias.

Note que nesses exemplos de regras de negócio a palavra "sistema" não aparece. Isso é comum, porque regras de negócio costumam valer dentro de um domínio (área de negócio), de forma independente do sistema a ser construído. Entretanto, é importante que aquelas regras que o sistema terá que satisfazer ou que afetem sua construção de alguma forma sejam especificadas.

Podemos sumarizar a discussão sobre tipos de requisitos com base na figura a seguir. É comum que requisitos sejam inicialmente descritos como requisitos do usuário, menos detalhados e mais voltados para os objetivos do usuário, e posteriormente detalhados como requisitos do sistema. Independente do nível de detalhamento, eles podem ser requisitos funcionais ou requisitos não funcionais do sistema em si ou requisitos de domínio da área para a qual o sistema está sendo construído.

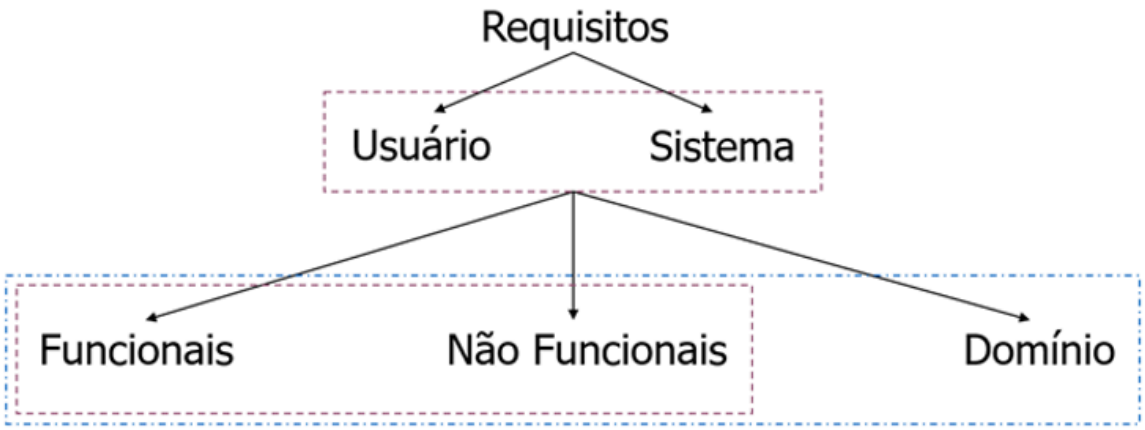


Figura 4.3: Tipos de requisitos.

4.3 Requisitos em contextos de transformação digital e ágeis

Um conceito comumente utilizado em contextos de transformação digital é o do MVP (***Minimum Viable Product***, ou **produto**

mínimo viável). O conceito do MVP se popularizou com a abordagem *Lean Startup*, que tem como princípio fundamental o uso do ciclo de aprendizado evolutivo *Build-Measure-Learn* (*Crie-Meça-Aprenda*) para transformar ideias em produtos, medir como os clientes respondem a estes produtos e então avaliar se a ideia deve ser levada adiante.

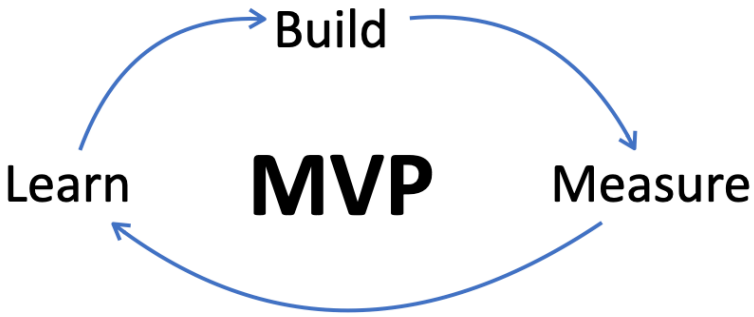


Figura 4.4: Aprendizado evolutivo de um MVP.

Um MVP pode ser definido como uma versão enxuta de um novo produto, que permite à equipe coletar o máximo de aprendizado validado com o menor esforço. Um MVP precisa permitir avaliar hipóteses de negócio, como avaliar se o uso de um recurso de inteligência computacional permitirá aumentar as vendas da empresa em x%.

No caso de produtos de software, os MVPs devem ser construídos de forma enxuta, mas pensando na sua possível evolução através de incrementos futuros. Na área de software, sabemos do elevado custo de evoluir soluções que não foram construídas com qualidade e que construir um MVP com qualidade em mente pode evitar grandes perdas com retrabalho futuro. Ou seja, **um MVP de software deve ser mínimo em escopo, não em qualidade** (ALONSO *et al.*, 2023).

Nos contextos de transformação digital, a elaboração do documento de visão é comumente substituída por dinâmicas de ideação e a geração de um **MVP Canvas**. Além disso, em contextos ágeis de

desenvolvimento, como visto no capitulo anterior, hoje predominantes no mercado, o documento de requisitos do sistema é comumente substituído por um ***backlog* do produto** (*product backlog*), em que os itens são **histórias de usuário** (*user stories*).

Em relação à especificação da funcionalidade em si, o MVP Canvas lista *features* identificadas durante as dinâmicas de ideação. ***Features*** são funcionalidades ou recursos que têm como propósito adicionar uma nova entrega de valor e experiência para seus usuários. O termo *feature* é utilizado de maneira similar ao conceito de um requisito funcional do usuário, descrevendo de forma abstrata uma funcionalidade que o software deve prover. Essas *features* são então detalhadas em histórias de usuário, normalmente pelo *Product Owner*. Essas histórias de usuário, por sua vez, serão quebradas em tarefas pela equipe de desenvolvimento. A figura a seguir mostra a estrutura desses agrupamentos. Um MVP tem features que são detalhadas em histórias de usuário, que por sua vez são detalhadas em tarefas.

Figura 4.5: Detalhamento de componentes ágeis.

A dinâmica de ideação é o ponto de partida e existem hoje diversas abordagens para conduzir essa dinâmica, tais como **Design Thinking**, **Design Sprints**, **Lean Startup** e **Lean Inception**.

Exemplificaremos essas dinâmicas através da **Lean Inception**, que é definida como a combinação do *Design Thinking* com *Lean Startup* para decidir o produto mínimo viável (MVP). Mais informações sobre essa dinâmica podem ser encontradas no livro escrito pelo criador dela, o Paulo Caroli (2018).

A figura a seguir ilustra a agenda típica de uma *Lean Inception*. É possível observar uma sequência de atividades sendo conduzidas em sessões com a participação dos diferentes interessados. Para cada uma dessas atividades há *templates* gratuitos disponíveis e elas estão descritas resumidamente a seguir.

Figura 4.6: Lean Inception - Agenda. Fonte: Caroli, 2018.

- ***Kick off***: A sessão de *kick off* serve para orientar os participantes em relação ao problema a ser resolvido, dando

início ao entendimento comum das principais necessidades e objetivos.

- **Visão do produto:** Aqui é construída uma visão comum para o produto, seguindo um *template* específico para isso. A visão é naturalmente inicial e será refinada durante a *Inception*.
- **Definição inicial de escopo:** Nesta sessão se definem os objetivos principais e se lista o que o produto é, o que ele não é, o que ele faz e o que ele não faz.
- **Personas:** Essa sessão tem foco nos usuários e seus objetivos. Na *Lean Inception*, usuários são representados através de *personas*, que, de forma simplificada, podem ser entendidas como grupos de usuários que compartilham as mesmas características, desejos e necessidades.
- ***Brainstorming* de *features*:** Aqui são levantadas as *features* candidatas para o produto. É recomendado que a descrição das *features* seja a mais simples possível. Como em um *brainstorm* tradicional, há uma fase de geração de ideias e uma fase de agregação e consenso.
- **Revisão técnica, de UX e de negócio:** Nesta atividade, cada *feature* é classificada de acordo com como equipe se sente em relação ao entendimento técnico e o entendimento de negócio. Cada *feature* é ainda avaliada em relação ao esforço, ao valor agregado para a experiência do usuário e para o negócio.
- **Jornadas:** Nesta atividade se descrevem as *jornadas dos usuários*, que são percursos de um usuário por uma sequência de passos para alcançar um objetivo. Alguns desses passos representam diferentes pontos de interação do usuário com o produto.
- **Sequenciador:** A partir do conjunto de *features* já levantadas, precisamos elaborar um plano de entrega de *features* que farão parte do MVP.
- **MVP Canvas:** Explicita o resultado final da visão do MVP, listando as hipóteses do negócio, as métricas para testar essas hipóteses, as *features*, as personas e jornadas atendidas, e o custo e cronograma para a entrega. É comum que o custo e

cronograma sejam apresentados no *MVP Canvas* de forma mais informal.

- ***Showcase*:** O MVP Canvas é apresentado. Essa é mais uma oportunidade para repensar se a visão do produto, as personas, as jornadas, as *features* e o MVP ainda fazem sentido.

Vejamos o resultado fictício de um MVP Canvas, seguindo o *template* da *Lean Inception* na figura a seguir, referente a um banco que busca reduzir a inadimplência dos seus clientes no pagamento dos seus empréstimos.

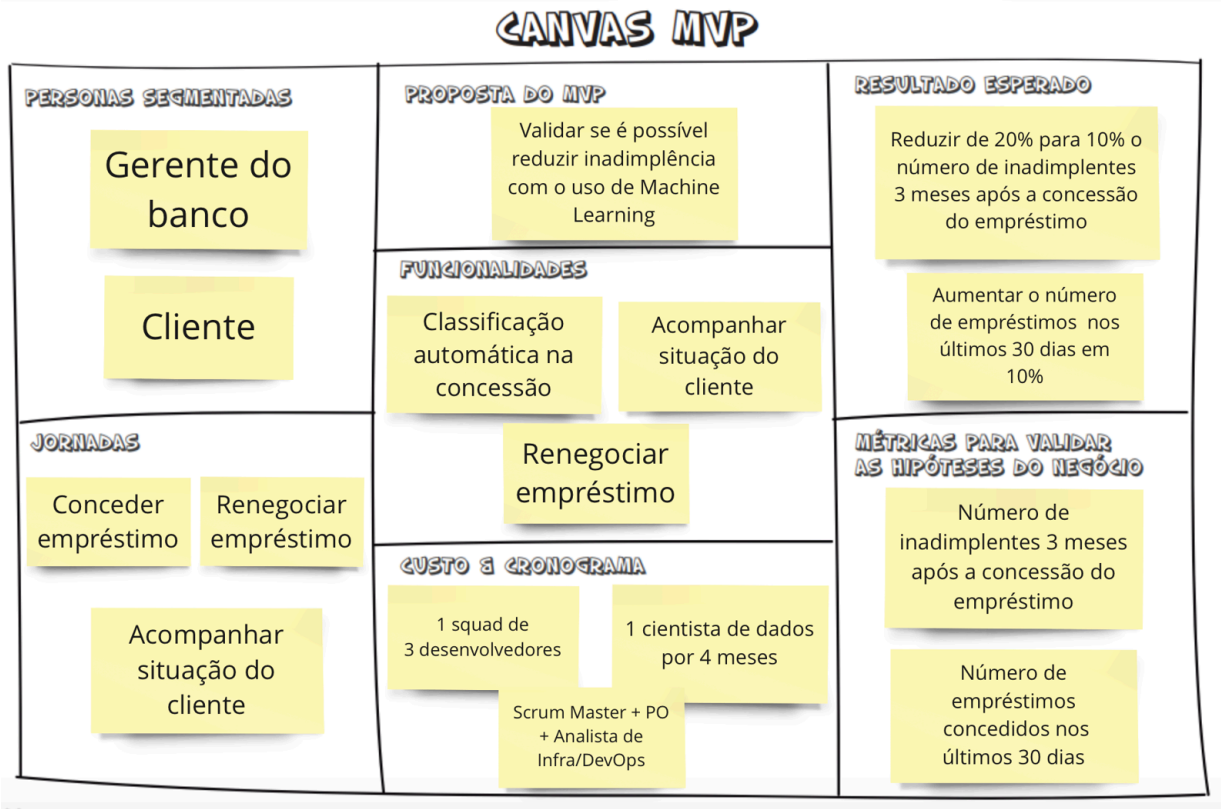


Figura 4.7: Exemplo de MVP Canvas.

Como pode ser visto, a proposta desse MVP é validar se é possível reduzir a inadimplência com o uso de Inteligência Computacional (neste caso, *Machine Learning*). É possível observar ainda que ele envolve três *features* principais: classificação automática na concessão, renegociar empréstimo e acompanhar a situação do cliente.

Uma vez definido o MVP, o próximo passo é quebrar suas *features* em **histórias do usuário**. Uma história de usuário é uma descrição simples e curta de uma funcionalidade contada pela perspectiva de uma pessoa que deseja desempenhar uma atividade no sistema. O formato padrão para descrever histórias de usuário é o "**Como** `<quem?>` **, quero** `<o quê?>` **para** `<por quê?>` ", onde:

- `<quem?>` define quem é o tipo de usuário (persona) que tem a necessidade;
- `<o quê?>` define o que o usuário deseja fazer. Ao se partir de um MVP Canvas, neste ponto pode entrar diretamente a *feature*, ou então, para *features* mais macro e abstratas, pode ser necessário quebrar uma *feature* em mais de uma história de usuário. Veremos exemplos de ambas as situações;
- `<por quê?>` define qual o benefício do usuário ao ter a funcionalidade desenvolvida para atender a essa necessidade. Em outras palavras, qual o valor direto obtido pelo usuário.

Veja o exemplo para a quebra de duas das *features* do MVP do nosso exemplo em histórias de usuário:

EXEMPLO:

Feature:

(F01) Classificação automática na concessão.

User story:

(US01) **Como** gerente do banco, **quero** realizar a classificação automática de um cliente **para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão.

Já a segunda *feature* está relacionada com a renegociação do empréstimo.

EXEMPLO:

Feature:

(F02) Renegociar empréstimo

User stories:

(US02) **Como** gerente do banco, **quero** consultar os dados de um empréstimo vigente **para** saber a situação dos pagamentos de um cliente.

(US03) **Como** gerente do banco, **quero** consultar condições de renegociação de valores e parcelas **para** saber o que posso oferecer ao cliente na renegociação.

(US04) **Como** gerente do banco, **quero** registrar a renegociação do empréstimo **para** que as novas condições de valores e parcelas fiquem vigentes.

Note que nesse exemplo uma *feature* virou uma única história de usuário, enquanto a outra foi desmembrada em três. É bastante comum que *features* referentes a aprendizado de máquina virem uma única história de usuário, como nesse exemplo. É evidente, entretanto, que essa especificação precisará ser complementada para subsidiar o trabalho do cientista de dados. Veremos como fazer isso na próxima seção.

Ao decidir sobre a quebra de uma *feature* em história de usuário, deve-se ter em mente as características de uma boa história de usuário, que podem ser memorizadas pelo acrônimo **INVEST**:

- **I** - Devem ser *Independentes*: dadas duas histórias X e Y, deve ser possível implementá-las em qualquer ordem;
- **N** - Devem ser abertas para *Negociação*: são convites para conversas entre clientes e desenvolvedores durante uma iteração;

- **V** - Devem agregar *Valor* para o negócio do cliente;
- **E** - Deve ser viável *Estimar* o tamanho de uma história;
- **S** - Devem ser *Sucintas* para facilitar o entendimento e a estimativa delas;
- **T** - Devem ser *Testáveis* com critérios de aceitação objetivos.

Além disso, histórias de usuário são comumente complementadas com descrições informais que envolvem a listagens de campos, regras de negócio que devem ser obedecidas, regras de validação (como campos obrigatórios, entre outras), protótipos de telas (*wireframe*, sem implementação) e aspectos não funcionais a serem considerados (como segurança ou usabilidade).

Outra prática muito comum em métodos ágeis é complementar a história de usuário com **critérios de aceitação** (também conhecidos como cenários BDD — *Behavior-Driven Development*). No contexto ágil, *definition of ready* (que significa que uma história está preparada para ser implementada) é comumente associado a ter detalhados a história de usuário e os seus critérios de aceitação. Já o *definition of done* (que significa que uma história foi implementada e está feita) é obtido quando uma história de usuário foi implementada e testada, atendendo aos critérios de aceitação.

Um *critério de aceitação* de uma história de usuário descreve um conjunto ordenado de comportamentos baseado em uma entrada para alcançar um resultado específico de uma funcionalidade. Costumam ser descrições adicionais utilizadas para a aceitação da história de usuário. O formato padrão para os critérios de aceitação é o "**Dado que** `<contexto?>` **quando** `<ação?>` **então** `<resultado?>` ", onde:

- `<contexto?>` descreve em qual ponto o cenário iniciará;
- `<ação?>` define a ação (ou evento) que está sendo realizada pelo usuário no contexto descrito;
- `<resultado?>` define o resultado esperado da ação.

Veja um exemplo para a *user story* US01 do exemplo anterior, referente à classificação automática de clientes em bons ou maus pagadores utilizando *Machine Learning*.

EXEMPLO:

User Story:

(US01) **Como** gerente do banco, **quero** realizar a classificação automática de um cliente **para** saber se o cliente pagará um empréstimo e decidir sobre a sua concessão.

Critérios de aceitação:

Dado que um bom pagador solicita um empréstimo, **Quando** consulto a classificação automática, **Então** o sistema me sugere conceder o empréstimo em pelo menos 90% dos casos.

Dado que um mau pagador solicita um empréstimo, **Quando** consulto a classificação automática, **Então** o sistema me sugere não conceder o empréstimo em pelo menos 90% dos casos.

Considerando esses cenários de aceitação, o sistema precisaria ser testado utilizando dados de bons e maus pagadores não vistos durante a construção do modelo. Para a avaliação, seria utilizada uma métrica bastante conhecida no meio de algoritmos de classificação, que é o *recall* (percentual corretamente recuperado) para bons e para maus pagadores. Se o *recall* para bons pagadores for acima de 90%, o sistema atenderá ao primeiro critério de aceitação. Se o *recall* para maus pagadores também for acima de 90%, o sistema atenderá aos dois critérios de aceitação.

As histórias de usuário (com seu detalhamento) ficam armazenadas no *backlog* do produto, que deve ser mantido e refinado pelo *Product Owner*, conforme vimos no capítulo anterior.

4.4 Requisitos de sistemas de software inteligentes

Como explicado anteriormente, a especificação baseada em histórias de usuário precisará ser complementada para subsidiar bem o trabalho da pessoa cientista de dados. De fato, a especificação é considerada a atividade mais desafiante do desenvolvimento de sistemas baseados em *Machine Learning*.

Para subsidiar a construção do modelo de *Machine Learning* e a sua integração com o sistema, a história de usuário precisaria ser complementada com: informações sobre o tipo de problema de *Machine Learning* sendo tratado (por exemplo, classificação, regressão, agrupamento ou associação); as informações de entrada disponíveis para o modelo; as saídas esperadas; as fontes de dados para o aprendizado; detalhes sobre a coleta dos dados; informações sobre a dinâmica de criação e atualização do modelo; sobre a dinâmica de execução do modelo; sobre como o modelo será avaliado antes de ser implantado (avaliação off-line); como o modelo será monitorado e o que deverá ser monitorado, além de outras características de qualidade desejadas (qualidade dos dados, explicabilidade, aspectos éticos, confiabilidade/acurácia do modelo, entre outros) e o grau em que se deseja cada uma.

Além disso, ainda é preciso pensar nos requisitos não funcionais e de infraestrutura referentes ao sistema em que o modelo será incluído. Resumindo, especificar sistemas baseados em *Machine Learning* é complexo.

Vejamos exemplos de problemas de classificação, regressão, agrupamento e associação no formato de histórias de usuário.

- **Classificação**
    - **Feature:** (F01) Classificação automática na concessão de bons e maus pagadores.

    - **User story:** (US01) Como gerente do banco, quero realizar a classificação automática de um cliente para saber se o cliente pagará um empréstimo e decidir sobre a sua concessão.
- **Regressão**
    - **Feature:** (F01) Predição do faturamento.
    - **User story:** (US01) Como diretor financeiro da empresa, quero saber o faturamento esperado para o próximo trimestre para apoiar o planejamento de investimentos.
- **Agrupamento**
    - **Feature:** (F01) Identificação de localidade para novas filiais.
    - **User story:** (US01) Como diretor de expansão, quero saber qual o local mais promissor para abrir uma nova filial para decidir sobre o local de abertura.
- **Associação**
    - **Feature:** (F01) Oferta de novos produtos
    - **User story:** (US01) Como cliente da loja *X*, quero ver produtos relacionados aos meus interesses para encontrar o produto ideal para minha necessidade.

Independente do tipo de problema de *Machine Learning*, o formato de uma história de usuário é claramente insuficiente para subsidiar a construção do modelo e a especificação precisa ser complementada. Note que normalmente uma *feature* de *Machine Learning* é mapeada em uma única história de usuário e que ela não provê nenhum detalhe para o(a) cientista de dados.

4.5 ML Canvas

Uma alternativa para especificar aspectos relacionados a uma solução de *Machine Learning* é o ***Machine Learning Canvas*** (ou

ML Canvas), proposto por Dorard (2015). O *Machine Learning Canvas* descreve como sistemas de *Machine Learning* tornarão predições em valor para os usuários finais, considerando elementos como a coleta de dados, monitoração do modelo e a proposição de valor. Essa é, provavelmente, a abordagem mais amplamente utilizada atualmente na prática para descrever sistemas que possuem componentes de *Machine Learning*, embora também tenha claras limitações do ponto de vista de Engenharia de Software.

O ML Canvas pode ser visto na figura a seguir. Por um lado ele é prático e captura informações essenciais para apoiar a construção ágil de algum modelo de *Machine Learning*. Por outro lado ele é desintegrado do processo de desenvolvimento e claramente incompleto do ponto de vista de Engenharia de Software, não capturando perspectivas e preocupações que podem ser muito relevantes do ponto de vista prático. Por exemplo, ele não captura preocupações referentes à infraestrutura em que o modelo será servido, à forma de interação com o usuário, à forma de visualização dos dados, ao tamanho do modelo, à explicabilidade, à privacidade, entre muitas outras.

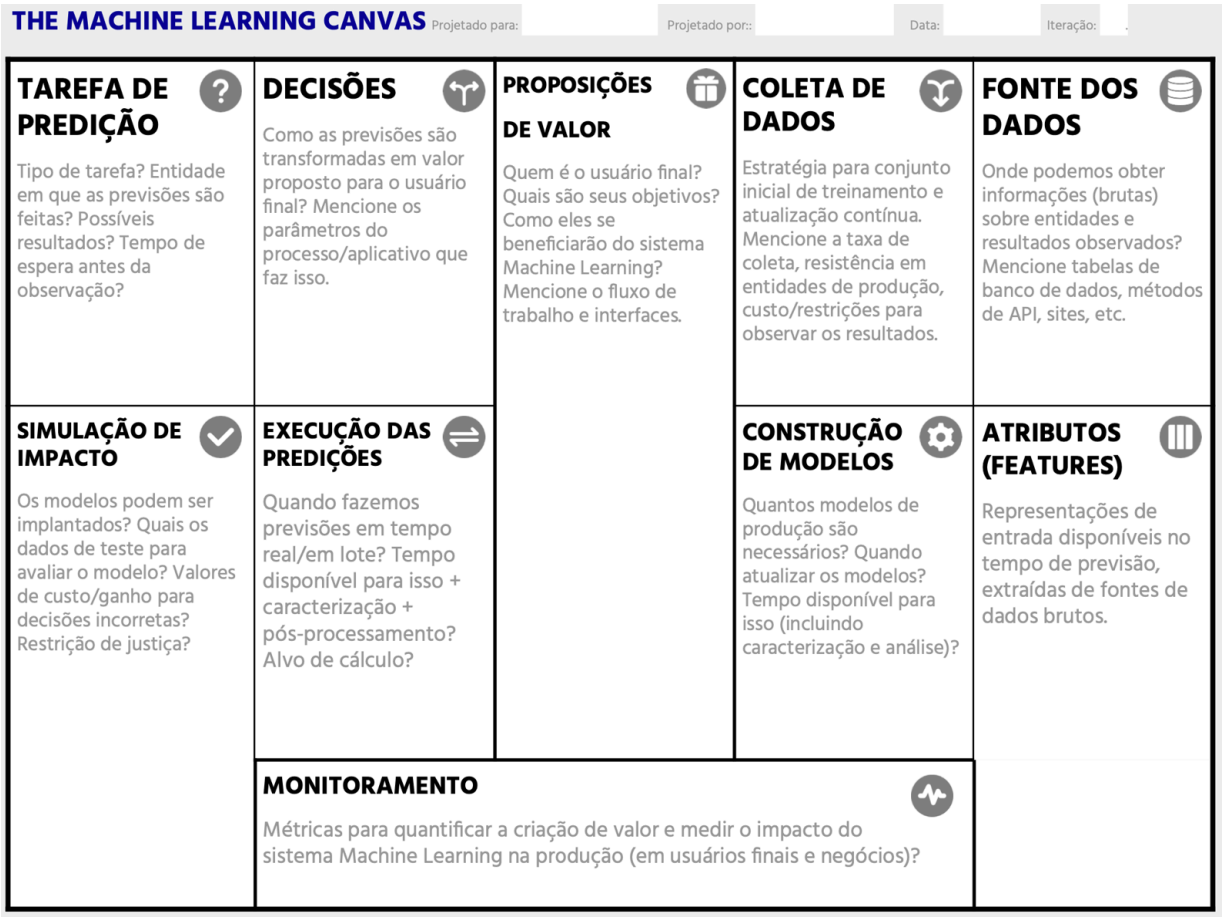


Figura 4.8: Machine Learning Canvas. Fonte: Dorard, 2015.

4.6 PerSpecML — Especificação baseada em perspectivas

A abordagem de especificação de *Machine Learning* baseada em perspectiva (*PerSpecML – Perspective-based Specification of Machine Learning-Enabled Systems*) se diferencia das demais por introduzir um conjunto de perspectivas e preocupações relevantes para a especificação de sistemas baseados em *Machine Learning* (VILLAMIZAR *et al.*, 2023). A abordagem foi avaliada junto a diversas empresas com resultados positivos e cobre uma visão mais ampla do que o ML Canvas, podendo ser considerada o estado da

arte para a especificação de sistemas baseados em *Machine Learning*.

As perspectivas e preocupações foram levantadas considerando preceitos da Engenharia de Software e um conjunto de avaliações. Elas servem como uma espécie de *checklist*, assegurando que a especificação considerou os aspectos mais importantes para subsidiar apropriadamente a construção de sistemas baseados em *Machine Learning*.

PerSpecML considera 5 perspectivas e 51 preocupações:

1. **Objetivos de *Machine Learning*** (7 preocupações a serem consideradas);
2. **Experiência do Usuário** (7 preocupações a serem consideradas);
3. **Infraestrutura** (10 preocupações a serem consideradas);
4. **Modelo** (11 preocupações a serem consideradas);
5. **Dados** (16 preocupações a serem consideradas).

1. Perspectiva de objetivos de *Machine Learning*

Fazer a ponte entre os objetivos de alto nível e as propriedades detalhadas dos modelos de *Machine Learning* é uma das causas mais comuns de fracasso desse tipo de projetos. É difícil de definir o sucesso em sistemas de *Machine Learning* com uma única métrica. Uma boa prática é definir o sucesso em diferentes níveis. Essa perspectiva compreende as seguintes preocupações:

Preocupação	Tratar essa preocupação envolve especificar ...
Problema e contexto	o problema de *Machine Learning* a ser abordado e seu contexto. *Machine Learning* deve ser direcionado ao problema certo.
Objetivos organizacionais	benefícios mensuráveis que o *Machine Learning* deve trazer para a organização. Por exemplo, aumentar a receita em x%, aumentar o número de unidades vendidas em *y%*.
Objetivos do usuário	o que os usuários desejam alcançar usando *Machine Learning*. Por exemplo, para sistemas de recomendação, isso pode envolver ajudar os usuários a encontrar conteúdo de que gostam.
Objetivos do modelo	métricas e medidas aceitáveis que o modelo deve alcançar (por exemplo, para problemas de classificação, isso pode envolver acurácia > x%; precisão > y%; *recall* > z%).
Funcionalidade de *Machine Learning*	a funcionalidade que o modelo fornecerá (por exemplo, classificar clientes, prever probabilidades de reclamações).
Principais indicadores	medidas correlacionadas com o sucesso futuro, a partir da perspectiva do negócio. Isso pode incluir os estados afetivos dos usuários ao usar o sistema (por exemplo, sentimento e engajamento do cliente).

Preocupação	Tratar essa preocupação envolve especificar ...
Expectativas do cliente	o equilíbrio das expectativas do cliente (por exemplo, tempo de inferência x precisão; falso positivo x falso negativo).

2. Perspectiva de experiência do usuário

Um bom sistema de *Machine Learning* inclui a criação de boas experiências do usuário. O objetivo dessa perspectiva é analisar como apresentar as previsões do modelo para os usuários e como coletar o *feedback* do usuário para melhorar o modelo.

Preocupação	Tratar essa preocupação envolve especificar ...
Valor	o valor agregado percebido pelos usuários das previsões para o seu trabalho.
Contundência (*Forcefulness*)	com que intensidade o sistema força o usuário a fazer o que o modelo indica que ele deve fazer (por exemplo, ações automáticas ou assistidas).
Frequência	a frequência com que o sistema interage com os usuários. Por exemplo, interagir sempre que o usuário solicitar ou sempre que o sistema achar que o usuário responderá.
Visualização	a maneira de apresentar os resultados de *Machine Learning* para que os usuários possam entendê-los (por exemplo, especificar *dashboards* e protótipos de visualização para validação).
Interatividade (*Interactiveness*)	quais interações os usuários terão com o sistema (por exemplo, para fornecer novos dados para aprendizado ou sistemas *human-in-the-loop*, em que os modelos exigem interação humana).
Custo	o impacto para o usuário de uma previsão errada do modelo.
Responsabilidade (*Accountability*)	quem é responsável por resultados inesperados ou ações tomadas com base em resultados do modelo.

3. Perspectiva de infraestrutura

Os modelos de *Machine Learning* produzidos por cientistas de dados normalmente são parte de sistemas conectados que exigem características especiais quando em operação. O objetivo dessa perspectiva é abranger a execução dos modelos, o monitoramento do modelo e seu aprendizado contínuo.

Preocupação	Tratar essa preocupação envolve especificar ...
Transmissão de Dados (*Data Streaming*)	qual estratégia de transmissão de dados será usada (por exemplo, transporte de dados em tempo real ou em lotes).
Fornecimento do modelo (*Model Serving*)	a maneira de executar e consumir o modelo (por exemplo, no *back-end* de uma aplicação, baseado em nuvem, web service).
Aprendizado Incremental	a necessidade de o modelo aprender continuamente com novos dados, ampliando o conhecimento do modelo existente.
Armazenamento	onde os artefatos de *Machine Learning* (por exemplo, modelos, dados, scripts) serão armazenados.
Monitorabilidade	a necessidade de monitorar os dados de operação e as saídas do modelo para alertar/detectar quando os dados flutuam ou mudam.
Telemetria	quais dados de uso dos sistemas de *Machine Learning* precisam ser coletados. A telemetria envolve a coleta de dados como cliques em botões específicos e pode envolver outros dados de uso.
Manutenibilidade	a necessidade de modificar o sistema inteligente para melhorar o desempenho ou se adaptar a um novo ambiente.
Reprodutibilidade	a necessidade de executar repetidamente um experimento ou algoritmo em determinados conjuntos de dados e obter

Preocupação	Tratar essa preocupação envolve especificar ...
	os mesmos resultados (ou resultados semelhantes).
Integração	a integração que o modelo terá com o restante da funcionalidade do sistema.
Custo	o custo financeiro envolvido com a infraestrutura que pode afetar decisões arquiteturais. Grandes modelos podem ser inutilizáveis devido ao custo para executá-los e mantê-los.

4. Perspectiva do modelo

Construir um modelo implica não apenas treinar um algoritmo de *Machine Learning* com dados para prever ou classificar bem algum fenômeno. Muitos outros aspectos da qualidade de um modelo podem ser importantes ao operar um sistema inteligente.

Preocupação	Tratar essa preocupação envolve especificar ...
Entradas e saídas	as entradas (*features*) e resultados esperados (*target*) do modelo. O conjunto de entradas significativas pode ser refinado/aprimorado durante atividades de pré-processamento, como *feature selection*.
Seleção do algoritmo e do modelo	o conjunto de algoritmos que podem ser usados/investigados, com base no problema e em outras preocupações a serem consideradas (restrições relacionadas à explicabilidade ou desempenho do modelo, por exemplo, podem limitar as opções de solução).
Métricas de desempenho	as métricas usadas para avaliar o desempenho do modelo (por exemplo, precisão, *recall*, *F1-score*, erro quadrático médio) e expectativas de desempenho mensuráveis.
Otimização de hiperparâmetros do algoritmo	a necessidade de escolher um conjunto de hiperparâmetros ótimos para um algoritmo de *Machine Learning*. Um hiperparâmetro é um parâmetro usado para controlar o processo de aprendizado.
Explicabilidade / Interpretabilidade	a necessidade de entender os motivos das inferências do modelo. O modelo pode precisar ser capaz de resumir as razões de suas decisões. Outras preocupações relacionadas, como transparência e interpretabilidade, podem também ser relevantes.

Preocupação	Tratar essa preocupação envolve especificar ...
Tempo de inferência	o tempo aceitável para executar o modelo e retornar as previsões.
Tempo de aprendizado	o tempo aceitável para treinar o modelo.
Tamanho do modelo	o tamanho do modelo em termos de armazenamento e sua complexidade (por exemplo, para árvores de decisão pode haver necessidade de poda).
Modelo-base	o modelo simples opcional que atua como referência. Sua principal função é contextualizar os resultados dos modelos treinados.
Degradação de desempenho	aspectos de degradação de desempenho. Com o tempo, o desempenho preditivo de muitos modelos diminui à medida que ele é testado em novos conjuntos de dados em ambientes em rápida evolução.

5. Perspectiva dos dados

Dados ruins resultarão em previsões imprecisas. Portanto, *Machine Learning* requer dados de entrada de alta qualidade. A perspectiva de dados foi construída com base no modelo de Qualidade de Dados definido na norma ISO/IEC 25012 e em experiências práticas.

Preocupação	Tratar essa preocupação envolve especificar ...
Fonte	de onde os dados serão obtidos.

Preocupação	Tratar essa preocupação envolve especificar ...
Seleção dos dados	o processo de determinar o tipo de dados apropriado e amostras adequadas para coletar dados.
Acurácia	a necessidade de obter dados corretos.
Viés	a necessidade de obter amostras justas de dados e distribuições representativas.
Completude	a necessidade de obter dados contendo observações suficientes de todas as situações em que o modelo vai operar.
Consistência	a necessidade de obter dados consistentes em um determinado contexto.
Credibilidade	a necessidade de obter dados verdadeiros que sejam críveis e compreensíveis pelos usuários.
Quantidade	a quantidade de dados esperada de acordo com o tipo de problema e a complexidade do algoritmo.
Dados reais	a necessidade de obter dados reais que representem o problema real.
Ética e privacidade	a necessidade de obter dados para evitar impactos adversos na sociedade (por exemplo, listando possíveis impactos adversos a serem evitados).
Operações nos dados	quais operações devem ser aplicadas nos dados (por exemplo, limpeza e rotulagem de dados).

Preocupação	Tratar essa preocupação envolve especificar ...
Distribuição dos dados	as distribuições de dados esperadas e como os dados serão divididos em dados de treinamento e teste.
Dicionário de dados	nomes, definições e atributos para elementos e modelos de dados.
Conjunto de dados "Golden" (*Golden Dataset*)	a necessidade de um conjunto de dados aprovado por um especialista de domínio que reflita o problema. É também empregado para monitorar outros dados adquiridos posteriormente.
Modelagem de dados	o que é necessário para converter os dados na representação do modelo.
Prontidão (*Timeliness*)	o tempo entre quando os dados são esperados e quando estão prontamente disponíveis para uso.

Diagrama de tarefas e preocupações de *Machine Learning* baseado em perspectiva

As 5 perspectivas e 51 preocupações são resumidas em um diagrama que é apresentado na figura a seguir. Esta imagem está disponível no repositório do livro (https://github.com/profkalinowski/livroescd), caso você queira consultar uma versão em tamanho maior.

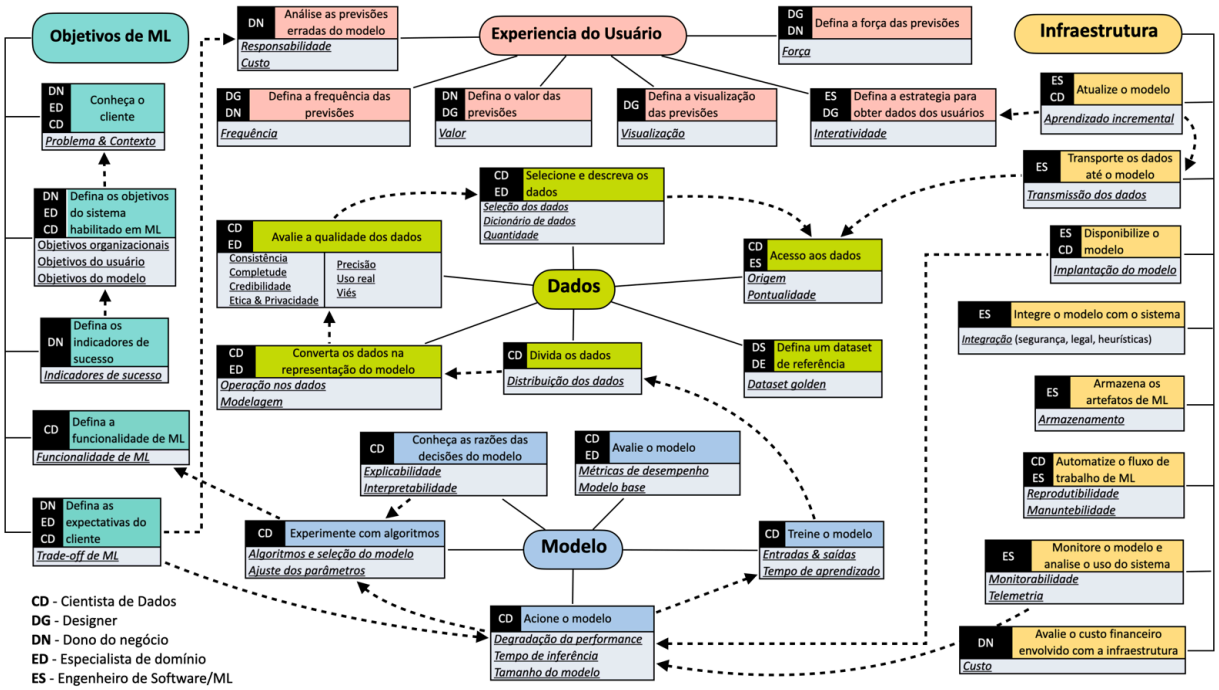


Figura 4.9: Diagrama de tarefas e preocupações de Machine Learning baseado em perspectiva.

O diagrama modela relações de dependência entre tarefas que são tipicamente alocadas em projetos de *Machine Learning* junto com as partes interessadas (*stakeholders*) que são tipicamente responsáveis pela execução e análise da tarefa. Essas tarefas possuem pelo menos uma preocupação que deveria ser analisada durante o processo de especificação do sistema de software inteligente.

Por exemplo, veja que normalmente em projetos de *Machine Learning*, cientistas de dados são responsáveis por treinar, executar e avaliar modelos. Essas tarefas envolvem preocupações implícitas que muitas vezes não são facilmente identificadas, como o tempo de inferência, o tamanho do modelo e os ajustes dos hiperparâmetros dos algoritmos de *Machine Learning*. Ao analisar o diagrama de tarefas e preocupações de *Machine Learning* baseado em perspectiva, a pessoa engenheira de requisitos, ou quem estiver a cargo desse papel, pode identificar, com o apoio das partes interessadas, quais tarefas e preocupações devem ser descritas como parte da especificação.

Após análise do diagrama de tarefas e preocupações de *Machine Learning* baseado em perspectiva, se uma preocupação for aplicável, deve ser especificada junto com as partes interessadas sugeridas usando um *template* de especificação alinhado com o diagrama. Para facilitar, o *template* está publicado e gratuitamente disponível no Miro, uma ferramenta colaborativa ágil que pode ser utilizada para realizar a especificação. O *template* pode ser acessado em https://miro.com/miroverse/perspecml/. Nele, você encontrará tudo que precisa para a aplicação da técnica, incluindo uma visão geral, um exemplo fictício de aplicação e um *template* vazio para que possa elaborar sua própria especificação. Como o Miro só permite a publicação de *templates* em inglês, uma alternativa é baixar o *template* em português e fazer o *upload* no Miro: https://github.com/dipucriodigital/ciencia-de-dados-e-analytics/blob/main/engenharia-de-sw-para-cd/PerSpecML-pt-BR.rtb.

Devido à natureza intensiva de comunicação e colaboração, bem como à interação inerente com a maioria dos outros processos de desenvolvimento de software, a Engenharia de Requisitos fornece a base para abordar vários dos desafios relacionados com a construção de sistemas de software inteligentes. Por exemplo, ao desenvolver modelos de *Machine Learning*, precisamos identificar dados relevantes e representativos, validar modelos e equilibrar as expectativas do usuário.

As técnicas apresentadas ao longo deste capítulo, em particular a PerSpecML, apoiam a engenharia de requisitos de sistemas inteligentes, fornecendo uma visão holística das perspectivas a serem consideradas e das preocupações que podem precisar ser detalhadas nesse tipo de projeto junto aos diferentes interessados.

Parte III – Programação com boas práticas de projeto e construção

Capítulo 5
Introdução à linguagem Python

A linguagem Python foi criada na década de 1980 por Guido Van Rossum com o intuito de ser uma linguagem de programação interpretada, mas que tivesse comandos simples e fáceis de entender, e cuja escrita fosse a mais próxima possível do inglês. Assim, a linguagem Python tem uma curva de aprendizado bem reduzida quando comparada com outras linguagens de programação, como Java ou C#.

5.1 Primeiro programa

Um programa em Python pode ser um único arquivo com a extensão `.py` ou uma pasta que contém várias subpastas com diversos arquivos contendo código Python e outras informações relevantes ao programa (como *datasets* e imagens).

Quando iniciamos o aprendizado de uma nova linguagem de programação, a forma mais comum de escrever o nosso primeiro programa é imprimir uma mensagem na tela, usualmente *"Hello, world!"*. No nosso primeiro programa, vamos imprimir uma mensagem na saída padrão (também denominada *console* ou linha de comando). Para imprimir uma mensagem na tela, basta utilizar a função `print()` com o texto a ser impresso na saída padrão dentro

dos parênteses e entre aspas. Em seguida, basta executar o código. Veja um exemplo a seguir:

```
# Isto é um comentário de apenas uma linha
print("Hello, world!")
```

Saída:

```
Hello, world!
```

Algumas observações:

- Lembre-se de que todas as vezes que abrir um parêntese, você deverá fechá-lo, assim como as aspas. Caso contrário, será exibida uma mensagem de erro pelo interpretador assim que você tentar executar o código.
- Em Python geralmente não finalizamos as instruções com uso de ponto e vírgula (é permitido, mas não é obrigatório). Quando existe a necessidade de colocar mais de uma instrução na mesma linha, é obrigatório o ponto e vírgula para informar ao interpretador o início e o fim de cada instrução.
- Os blocos de código em Python são delimitados por recuos (por padrão, 4 espaços). O espaço em branco é ignorado quando aparece dentro de parênteses e colchetes, o que é útil para a legibilidade.

Veremos a seguir alguns conceitos básicos importantes da linguagem Python.

5.2 Variáveis

Variáveis armazenam valores de determinados tipos para serem manipuladas em nossos programas. Em Python, não existe um

comando para declarar uma variável, ela é criada no momento em que atribuímos um valor a ela. Também não precisamos declarar os tipos das variáveis, podendo, inclusive, convertê-las para outro tipo posteriormente através de um *casting*. Os principais tipos de variáveis simples em Python são `int` (inteiro), `float` (real), `bool` (booleano, podendo assumir os valores `True` ou `False`) ou `str` (*string*, ou texto). Vejamos alguns exemplos:

```
# Variáveis: Definição e tipos
# Declarando 4 variáveis e atribuindo valores a elas
uma_string = "Aluno"
um_inteiro = 7
um_float = 7.584
um_booleano = True
# Imprimindo o valor e o tipo de uma das variáveis
print(um_float)
print(type(um_float))
```

Saída:

```
7.584
<class 'float'>
```

```
# Casting: Convertendo a variável do tipo float para int e para string
print(int(um_float))
print(str(um_float))
```

Saída:

```
7
7.584
```

Podemos trabalhar com operações matemáticas em Python, usando os operadores básicos soma (`+`), subtração (`-`), multiplicação (`*`), divisão (`\`), potenciação (`**`) e resto da divisão (`%`). Vejamos alguns exemplos:

```
# Operações Aritméticas

# Soma, Subtração, Multiplicação e Divisão
```

```
resultado_1 = 7 + 5
resultado_2 = um_inteiro - resultado_1
resultado_3 = resultado_2 * 3
resultado_4 = resultado_3 / 4

# Imprimindo os resultados concatenando uma string com o resultado
numérico
print("Resultado da soma = " + str(resultado_1))
print("Resultado da subtração = " + str(resultado_2))
print("Resultado da multiplicação = " + str(resultado_3))
print("Resultado da divisão = " + str(resultado_4))

# Potenciação e Resto da Divisão
resultado_5 = 2 ** 4
resultado_6 = 13 % 4

# Imprimindo os resultados concatenando uma string com o resultado
numérico
print("Resultado da potenciação = " + str(resultado_5))
print("Resultado do resto da divisão = " + str(resultado_6))
```

Saída:

```
Resultado da soma = 12
Resultado da subtração = -5
Resultado da multiplicação = -15
Resultado da divisão = -3.75
Resultado da potenciação = 16
Resultado do resto da divisão = 1
```

Também são muito utilizadas as operações de decremento (`-=`) e incremento (`+=`) onde, após o operador, especificamos a quantidade a decrementar ou incrementar. Vejamos alguns exemplos:

```
# Incremento e Decremento

# Decrementando 3 unidades
um_inteiro -= 3
print("Resultado: " + str(um_inteiro)) # 7 - 3 = 4 --> agora um_inteiro é
4
```

```
# Incrementando 5 unidades
um_inteiro += 5
print("Resultado: " + str(um_inteiro)) # 4 + 5 = 9 --> agora um_inteiro é
9
```

Saída:

```
Resultado: 4
Resultado: 9
```

Os operadores de igualdade permitem comparar se duas variáveis ou valores são iguais (`==`) ou diferentes (`!=`). Os operadores de comparação (`>` , `<` , `>=` , `<=`) também são muito utilizados para comparar variáveis numéricas. Também temos os operadores lógicos `and` (e) e `or` (ou). As possíveis respostas para os operadores de igualdade e comparação são os valores booleanos `True` (verdadeiro) ou `False` (falso). Vejamos alguns exemplos:

```
# Operadores de Igualdade

variavel_um = 1
variavel_dois = 2
variavel_tres = 3

print(1 == variavel_um)
print(2 == variavel_um)

equal = (variavel_um == variavel_dois) # os () são opcionais neste caso,
mas ajudam na legibilidade
not_equal = (variavel_um != variavel_dois)

print("Um é igual a dois? " + str(equal))
print("Um é diferente de dois? " + str(not_equal))

# Operadores de Comparação

print("Um é maior do que dois? " + str(variavel_um > variavel_dois))
print("Um é menor do que dois? " + str(variavel_um < variavel_dois))
```

Saída:

```
True
False
Um é igual a dois? False
Um é diferente de dois? True
Um é maior do que dois? False
Um é menor do que dois? True
```

```
# Operadores Lógicos and (e) e or (ou)

print((2 > 1) and (3 < 5)) # V V
print((2 > 1) and (1 > 3)) # V F

print((3 < 5) or (2 > 1)) # V V
print((0 > 1) or (0 == -1)) # F F
print((0 < 1) or (0 == -1)) # V F
```

Saída:

```
True
False
True
False
True
```

5.3 Strings

Strings armazenam uma ou mais palavras em texto. Existem diversas operações úteis com *strings*, tais como a concatenação de duas ou mais *strings*, verificação do tamanho de uma *string*, operações envolvendo indexação e *substrings*, multiplicação de uma *string* por um número, operador `in` , *escaping*, conversão para maiúsculas e minúsculas e formatação. Vejamos alguns exemplos:

```
# Concatenação
```

```
programacao = "Programação" # pode ser tanto delimitada por aspas
duplas...
oo = 'OO' #... quanto por aspas simples, mas sempre combinando
programacao_oo = programacao + " " + oo
print(programacao_oo)
```

Saída:

```
Programação OO
```

```
# Tamanho

print(len(programacao_oo))
```

Saída:

```
14
```

```
# Indexação e Substrings

frase = "Python é muito divertido!"

# A indexação começa com 0
print(frase[3])

# use -1 para pegar a última letra
print(frase[-1])

# -2 pega a penúltima letra
print(frase[-2])

# string[start:end] - end não é incluso
print(frase[:4])
print(frase[10:])
print(frase[5:10])
print(frase[:])
```

Saída:

```
h
!
```

```
o
Pyth
uito divertido!
n é m
Python é muito divertido!
```

```
# Multiplicação de uma string por um número

ola = "olá"
sete_olas = ola * 7
print(sete_olas)
```

Saída:

```
oláoláoláoláoláoláolá
```

```
# Operador in

programacao_python = "Programação em Python"

# checa se a string contém "Python"
print("Python" in programacao_python)

# checa se a string contém "python"
print("python" in programacao_python)

# checa se a string contém "abacate"
print("abacate" in programacao_python)
```

Saída:

```
True
False
False
```

```
# Escaping

print("Estou estudando\n" + programacao_python) # \n = nova linha
```

Saída:

```
Estou estudando
Programação em Python

# Maiúsculas e Minúsculas

print(programacao_python.lower()) # todas minúsculas
print(programacao_python.upper()) # todas maiúsculas

tudo_minuscula = "aula de python"
print(tudo_minuscula.capitalize()) # primeira letra maiúscula
```

Saída:

```
programação em python
PROGRAMAÇÃO EM PYTHON
Aula de python

# Formatação

nome = "Belinha"
numero_inteiro = 7
numero_decimal = 7.584

print("Olá, meu nome é %s! Tenho %d anos e %d reais" % (nome,
numero_inteiro, numero_decimal)) # float impresso como int
print("Olá, meu nome é %s! Tenho %d anos e %f reais" % (nome,
numero_inteiro, numero_decimal)) # float impresso como float
print("Olá, meu nome é %s! Tenho %d anos e %.2f reais" % (nome,
numero_inteiro, numero_decimal)) # float impresso como float com 2 casas
decimais
```

Saída:

```
Olá, meu nome é Belinha! Tenho 7 anos e 7 reais
Olá, meu nome é Belinha! Tenho 7 anos e 7.584000 reais
Olá, meu nome é Belinha! Tenho 7 anos e 7.58 reais
```

5.4 Coleções

Além dos tipos simples, como `int` e `float`, o Python nos fornece algumas estruturas de dados compostas ou que contenham outros tipos de dados, chamadas estruturas contêiner. São exemplos dessas estruturas as *listas*, as *tuplas* e os *dicionários*.

As **listas** são similares aos *arrays* e são definidas por `[]`. São sequências mutáveis e ordenadas, normalmente usadas para armazenar coleções de itens. As **tuplas** são definidas por `()`. São listas imutáveis, não sendo possível adicionar, alterar ou deletar itens de tuplas. Os **dicionários** são definidos por `{}`. São coleções desordenadas, mutáveis e indexadas com chaves e valores. As chaves de um dicionário precisam ser únicas e de tipos imutáveis como *strings*, números ou tuplas; já os valores de um dicionário podem ser de qualquer tipo. Vejamos alguns exemplos a seguir:

```
# Listas

numeros = [1, 10, 100, 1000]
print(numeros)

# imprimir o elemento de índice 2 - inicia no 0
print(numeros[2])
```

Saída:

```
[1, 10, 100, 1000]
100
```

```
# Operações de Listas

# incluir 2 itens na lista usando +=
numeros += [10000, 100000]
print(numeros)

# incluir 1 item na lista usando append
numeros.append(1000000)
print(numeros)
```

```
# substituir os itens nas posições 1 e 2 por 7
# [índice_incluído:índice_excluído]
numeros[1:3] = [7]
print(numeros)

# remover os itens nas posições 1 e 2 da lista
numeros[1:3] = []
print(numeros)

# tamanho da lista
print(len(numeros))
```

Saída:

```
[1, 10000, 100000, 1000000, 10000, 100000]
[1, 10000, 100000, 1000000, 10000, 100000, 1000000]
[1, 7, 1000000, 10000, 100000, 1000000]
[1, 10000, 100000, 1000000]
4
```

```
# Tuplas

naipes = ('copas', 'ouros', 'espadas', 'paus')
print(naipes)
```

Saída:

```
('copas', 'ouros', 'espadas', 'paus')
```

```
# Dicionários

# criar e imprimir um dicionário
notas = {"Ana": 8, "Maria": 5, "Thais": 10}
print(notas)

# acessar o valor correspondente à chave "Thais"
print(notas["Thais"])

# incluir novo item
notas["Zaira"] = 9
print(notas)
```

```
# remover item
del notas["Thais"]
print(notas)

# checar se notas contém o item "Maria"
print("Maria" in notas)
```

Saída:

```
{'Ana': 8, 'Maria': 5, 'Thais': 10}
10
{'Ana': 8, 'Maria': 5, 'Thais': 10, 'Zaira': 9}
{'Ana': 8, 'Maria': 5, 'Zaira': 9}
True
```

5.5 Condições

Em programação, é muito comum querermos executar um comando apenas se determinada condição for atendida. Para tal, temos os operadores `if` (se), `else` (senão) e `elif` (senão-se), que sempre retornam um resultado booleano (`True` ou `False`) e podem ser utilizados em conjunto com os operadores de igualdade, lógicos e de comparação. Vejamos alguns exemplos a seguir:

```
nome = "Ana"
idade = 25

# Comando if (condição se)

if nome == "Ana": # V
    print("Passou!")

if nome == "Ana" or idade == 17:   # V ou V = V
    print("Passou de novo!")
    print("%s tem %d anos" %(nome, idade))
```

Saída:

```
Passou!
Passou de novo!
Ana tem 25 anos
```

```
# Comandos if, elif e else

dias = ["sábado", "domingo"]
if len(dias) == 0:
    print("Lista vazia")
elif len(dias) == 1:
    print("Só um dia")
else:
    print("Tamanho %d" % len(dias))
```

Saída:

```
Tamanho 2
```

5.6 Loops

Quando queremos executar um comando até que ou enquanto determinada condição seja atendida, podemos usar os operadores `for` e `while`, que permitem fazermos um *loop*. Podemos, adicionalmente, utilizar os comandos `break` e `continue`. Vejamos alguns exemplos a seguir:

```
# Loop for
for i in range(6):
    print(i)

# OBS: A função range() pode ser representada de três formas diferentes:
# range(stop_value) : Considera o ponto inicial como zero.
# range(start_value, stop_value) : Gera a sequência com base no valor
inicial e final.
# range(start_value, stop_value, step_size): Gera a sequência
```

```
incrementando o valor inicial usando o tamanho do passo até atingir o
valor final.
```

Saída:

```
0
1
2
3
4
5
```

```
# Loop for - outro exemplo
pares = [2, 4, 6, 8]
for i in range(len(pares)):
    print("Índice %d - Valor %d " % (i, pares[i]))
```

Saída:

```
Índice 0 - Valor 2
Índice 1 - Valor 4
Índice 2 - Valor 6
Índice 3 - Valor 8
```

```
# Loop for usando string

hello_world = "Hello, World!"

# imprimir cada caractere
for ch in hello_world:
    print(ch)

# contar a quantidade de caracteres usando loop
length = 0
for ch in hello_world:
    length += 1
print("Número de caracteres: %d " % length)
```

Saída:

```
H
e
l
l
o
,

W
o
r
l
d
!
Número de caracteres: 13
```

```python
# Loop While
numero = 1

# imprimir de 1 a 7
while numero <= 7:
    print(numero)
    numero += 1
```

Saída:

```
1
2
3
4
5
6
7
```

```python
# imprimir os quadrados menores que 25
numero = 1
quadrado = 1

while quadrado <= 24:
    quadrado = numero ** 2
    print("Número %d - Quadrado %d" % (numero, quadrado))
```

```
    numero += 1

print("Ao sair do while, numero tem o valor %d" % numero)
```

Saída:

```
Número 1 - Quadrado 1
Número 2 - Quadrado 4
Número 3 - Quadrado 9
Número 4 - Quadrado 16
Número 5 - Quadrado 25
Ao sair do while, numero tem o valor 6
```

```
# Break

count = 0

# loop infinito - nunca é False
while True:
    print(count)
    count += 1
    if count >= 5:
      break # sai do loop
```

Saída:

```
0
1
2
3
4
```

```
naipes = ['copas', 'ouros', 'espadas', 'paus']

# loop infinito - nunca é False
while True:
    # pop(): retira e retorna o último item da lista
    elemento = naipes.pop()
    print(elemento)
    if elemento == "espadas":
        break
```

Saída:

```
paus
espadas
```

```
# Continue

for i in range(5):
    if i == 3:
       continue # pula o resto do código e move para a próxima iteração do
loop
    print(i)
```

Saída:

```
0
1
2
4
```

```
# imprimir somente os ímpares
for x in range(10):
    if (x % 2) == 0: # é par
        # pula print(x) para este loop
        continue
    print("Número %d" % x)
```

Saída:

```
Número 1
Número 3
Número 5
Número 7
Número 9
```

5.7 Funções

As funções possibilitam à pessoa desenvolvedora concentrar em um único lugar um trecho de código que pode ser chamado várias vezes ao longo do programa, o que facilita a manutenção da aplicação. As funções podem retornar algo e/ou receberem argumentos quando chamadas, bem como declararem parâmetros *default* (padrão). Vejamos alguns exemplos comentados a seguir:

```
# Funções

# definir uma função chamada hello_world
def hello_world():
    print("Hello, World!")
    print("Oi, Mundo!")
    print("Salut, Monde!")
    print("Hola, Mundo!")

# chamar a função 3 vezes
for i in range(3):
    hello_world()
```

Saída:

```
Hello, World!
Oi, Mundo!
Salut, Monde!
Hola, Mundo!
Hello, World!
Oi, Mundo!
Salut, Monde!
Hola, Mundo!
Hello, World!
Oi, Mundo!
Salut, Monde!
Hola, Mundo!
```

```
# Funções com parâmetros

# x é um parâmetro
def uma_funcao(x):
    print("x = %d" % x)
```

```
# passar 5 para a função. Aqui, 5 é um argumento passado para a função
uma_funcao(5)
```

Saída:

```
x = 5

# Funções com retorno

# função que retorna a soma de dois números
def soma(a, b):
    return a + b

c = soma(7, 5)
print("c = %d" % c)
```

Saída:

```
c = 12

# Função com parâmetros default (padrão)

def multiplica(a, b=2):
    return a * b

print(multiplica(5, 84))

# como b tem um valor padrão, podemos passar apenas um argumento
print(multiplica(7))
```

Saída:

```
420
14
```

5.8 Leitura e escrita de arquivos

Vejamos no exemplo a seguir como ler e escrever arquivos em Python:

```
naipes = ['copas', 'ouros', 'espadas', 'paus']

# Escrita de um arquivo

# Abrir o arquivo em modo de escrita ("w" para escrita)
f = open("naipes.txt", "w")

# escrever no arquivo
for i in naipes:
    f.write(i + "\n")

# fechar o arquivo
f.close()

# Leitura de um arquivo

# Abrir o arquivo em modo de leitura ("r" para leitura)
f = open("naipes.txt", "r")

# ler apenas a primeira linha
for line in f.readlines():
    print(line)

# fechar o arquivo
f.close()
```

Saída:

```
copas
ouros
espadas
paus
```

5.9 Conexão com banco de dados

Alguns recursos do Python não são carregados por padrão e precisam ser importados com o comando `import`. O módulo importado precisará ser referenciado para acessar as suas funções, sendo possível usar um *alias* para identificá-lo. Por exemplo:

```
import matplotlib.pyplot as plt

plt.plot(...)
```

Veremos mais exemplos ao longo do livro. Vejamos no exemplo a seguir como utilizar o banco de dados SQLite (que permite a utilização do banco de dados embutido no programa, como um arquivo) com Python:

```
# importação da biblioteca do SQLite
import sqlite3 # importando sem alias

# criar conexão (criará ou abrirá o arquivo .db do banco de dados)
con = sqlite3.connect("alunos.db")

# o cursor permite percorrer os registros de um banco de dados
c = con.cursor()

# criar uma tabela
c.execute("create table if not exists aluno(nome text, idade text)")

# inserir registros na tabela
c.execute("insert into aluno (nome, idade) values ('Ana', 25)")
c.execute("insert into aluno (nome, idade) values ('Maria', 38)")

# fazer o commit
con.commit()

# executar select * na tabela
```

```
result = c.execute("select * from aluno")

# imprimir a primeira linha do select
#print(result.fetchone())

# imprimir TODAS as linhas da tabela
print(result.fetchall())

# fechar a conexão
con.close()
```

Saída:

```
[('Ana', '25'), ('Maria', '38')]
```

Agora, execute esses trechos de código novamente fazendo pequenas modificações nos códigos, para praticar os conhecimentos adquiridos. Você pode acessar o código-fonte desses exemplos no link:

https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_intro_python.ipynb.

Bons estudos!

CAPÍTULO 6
Orientação a Objetos em Python

A **Programação Orientada a Objetos (POO)** é um paradigma de programação que faz uma aproximação para a visão de mundo que temos, na qual trabalhamos com transformação, manipulação e utilização de objetos. Um programa OO é estruturado como um conjunto de objetos que colaboram entre si para realizar as ações do sistema. Cada objeto provê serviços ou efetua ações e é usado por outros objetos. As principais estruturas de um programa OO são classes e objetos, e as ações são realizadas por meio da troca de mensagens entre objetos.

6.1 Classes e objetos

Em POO, as *classes* estão para *objetos* assim como formas estão para bolos: podemos preparar vários bolos a partir de uma mesma forma, assim como podemos criar vários objetos a partir de uma mesma classe, mas cada objeto terá sua própria identidade.

Uma **classe** é uma estrutura que integra dados e comportamento, e representa o conjunto de operações válidas que acessam e manipulam os seus respectivos valores. A classe é usada para construir objetos. Ela é o *projeto dos objetos* e fornece a descrição do comportamento (operações, representadas pelas *funções* ou *métodos*) comum e do estado (dados, representados pelas *variáveis*) dos objetos.

Já um **objeto** é uma instância de uma classe. Ele tem *estado* (seus dados, o que o objeto sabe sobre si mesmo) e *comportamento* (o que o objeto faz). Os objetos construídos a partir de uma classe compartilham comportamento comum a partir de uma mesma

classe, sendo "instância" um sinônimo de "objeto". O **comportamento** de um objeto é definido pela sua respectiva classe por meio de *métodos*, um conjunto de rotinas que operam sobre os dados. O **estado** de um objeto é definido pela sua respectiva classe por meio de *atributos*, que representam os dados de um objeto.

Por exemplo, você pode imaginar um objeto como um contato do seu celular. A entrada vazia pode ser considerada uma classe, e ela tem campos em branco (atributos). Quando você preenche os dados de um contato, está criando uma instância (objeto). Os métodos da classe são as operações que você pode fazer sobre uma entrada específica, tais como *consultar_telefone* e *editar_nome*.

Figura 6.1: Um novo contato em seu celular.

Como vantagens da Orientação a Objetos, podemos destacar o fato de os objetos serem representados de forma clara, com responsabilidades bem definidas e independentes dos outros objetos. Isso permite um código mais claro, de fácil manutenção, e

que trechos de códigos sejam facilmente reutilizados em outros sistemas. Os quatro pilares da Orientação a Objetos são:

1. **Encapsulamento**: funciona como uma "embalagem" ao redor do objeto, protegendo-o do acesso indevido por outros objetos;
2. **Abstração**: permite projetar a classe pensando nos comportamentos e estados que queremos que os objetos que serão criados a partir dela tenham (programar usando um nível de pensamento mais abstrato);
3. **Herança**: permite que as características e comportamentos descritos para caracterizar uma classe "pai" (superclasse) possam ser herdados pela sua classe "filha" (subclasse);
4. **Polimorfismo**: permite que métodos com a mesma semântica tenham múltiplas implementações.

Detalharemos esses pilares ainda neste capítulo, com exemplos de códigos para facilitar o entendimento dos conceitos.

Vimos que classe é uma estrutura de dados que une variáveis (propriedades) e funções (comportamentos). Em Python, tudo deriva de uma classe, inclusive tipos básicos como tipos numéricos (por exemplo, `int`) e textos (por exemplo, `str`). Veja o que acontece quando executamos o comando `help(int)`:

```
>>> help(int)
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
```

Figura 6.2: Trecho da execução do comando help(int).

Repare que o tipo `int` deriva da classe `int` que, por sua vez, herda da classe `object` os atributos e métodos comuns a todas as classes.

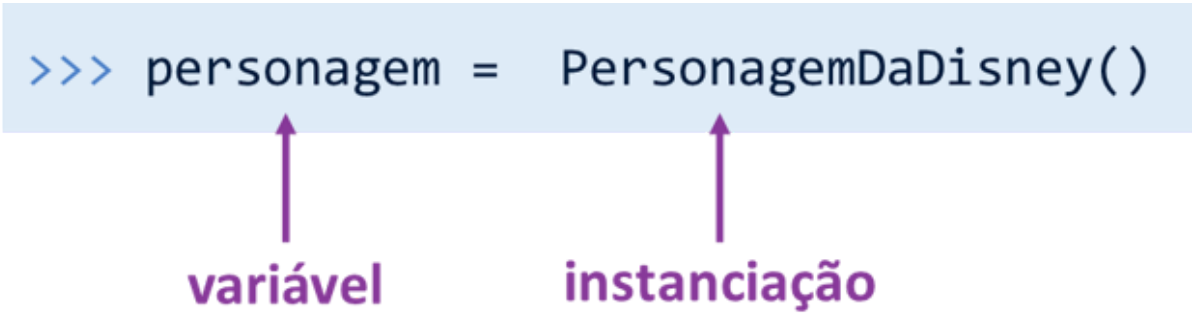
Em Python, classes podem ser definidas em qualquer lugar, inclusive dentro de funções. Usualmente, classes são definidas mais próximas do início do programa ou em módulos próprios, o que costuma deixar o código mais limpo e fácil de manter. A sintaxe para definir uma classe é:

```
class NomeDaClasse:
    # DECLARAÇÕES E COMANDOS
    pass
```

A palavra-chave `class` é usada para definir uma classe, e recomenda-se que seu nome utilize o padrão *CamelCase* (a separação das palavras é feita sem espaços e cada palavra inicia com letra maiúscula seguida de letras minúsculas). Por exemplo, suponha que estamos escrevendo o código-fonte de um jogo infantil. Vamos começar escrevendo uma classe para definir um personagem:

```
class PersonagemDaDisney:
    pass
```

Quando essa classe é usada para criar um personagem específico, estamos instanciando um objeto. Quando esse novo objeto é atribuído a uma variável, diz-se que essa variável referencia esse objeto:



6.2 Métodos

Dentro de classes é possível definir **métodos**, que representam as ações que os objetos criados a partir dessa classe poderão realizar. Um método recebe pelo menos um parâmetro (a instância da classe, o parâmetro `self`), que lhe permite saber a qual instância se referem aqueles atributos. Na declaração do método é preciso declarar `self` explicitamente, mas ao chamá-lo em um objeto, `self` é passado automaticamente.

Observação: o parâmetro `self` pode ter outro nome, mas `self` é uma convenção (padrão *PEP8*). Métodos podem receber qualquer quantidade de parâmetros, que podem ser obrigatórios ou não (declarados com um valor *default*).

Por exemplo, definimos uma classe `Pessoa` que possui dois métodos:

- `__init__`, inicializa o atributo `nome` durante a criação da instância;

- `__str__`, converte uma `Pessoa` para *string*. Observação: quando passamos um objeto para o comando `print`, ele vai buscar pelo método `__str__` para saber como representar o objeto na saída padrão.

```
class Pessoa:

    def __init__(self, nome):
        self.nome = nome

    def __str__(self):
        return self.nome


pessoa_1 = Pessoa("João")
print(pessoa_1)

pessoa_2 = Pessoa("Maria")
print(pessoa_2)
```

Saída:

```
João
Maria
```

As variáveis `pessoa_1` e `pessoa_2` referenciam instâncias (objetos) da classe `Pessoa`, com valores distintos para o atributo `nome`.

Para chamar um método em um objeto, basta escrever o nome da variável que o referencia e o nome do método separados por um ponto, e passar os parâmetros cabíveis. No exemplo, estamos chamando os métodos `soma` e `media` no objeto referenciado por `minha_lista`:

```
class ListaDeNumeros:

    def __init__(self, numeros):
        self.numeros = numeros

    def __str__(self):
```

```
        return ",".join([str(n) for n in self.numeros])

    def soma(self):
        return sum(self.numeros)

    def media(self):
        return self.soma()/len(self.numeros)

minha_lista = ListaDeNumeros([1,3,5,7,9])
print(f"A soma de {minha_lista} é {minha_lista.soma()}")
print(f"A média de {minha_lista} é {minha_lista.media()}")
```

Saída:

```
A soma de 1,3,5,7,9 é 25
A média de 1,3,5,7,9 é 5.0
```

6.3 Atributos

Existem três tipos de variáveis que podem ser declaradas dentro de uma classe: *atributos de classes*, *atributos de instâncias* e *variáveis de métodos*.

- **Atributos de classes** são atributos ligados à classe, e todas as instâncias da classe compartilham o mesmo atributo;
- **Atributos de instância** são específicos para cada instância, podendo ter valores diferentes de uma instância para outra;
- **Variáveis de métodos** existem apenas enquanto a função está sendo executada.

Exemplo: criamos a classe `Pessoa` com o atributo de classe `num_pessoas`, utilizado para contar quantas instâncias de `Pessoa` foram criadas. No método `__init__` criamos um atributo de instância `nome` para guardar o nome de cada indivíduo criado e incrementamos em uma unidade o atributo de classe `num_pessoas`:

```
class Pessoa:

    num_pessoas = 0

    def __init__(self, nome):
        self.nome = nome
        Pessoa.num_pessoas += 1
```

Cada instância de `Pessoa` terá um valor distinto de `nome`, mas o valor de `num_pessoas` será o mesmo em todas, uma vez que atributos de classe são compartilhados entre todas as instâncias.

Exemplo: criamos uma lista com quatro nomes e utilizamos uma *list comprehension* (recurso para trabalhar sobre dados existentes em objetos iteráveis, como listas) para inicializar quatro instâncias da classe `Pessoa` com cada um desses nomes. Depois, imprimimos:

- o valor de `Pessoas.num_pessoas` (note que o atributo de classe é acessado fazendo referência ao nome da classe); e
- o valor do nome da instância localizada na primeira posição da lista (note que o atributo de instância é acessado fazendo referência à variável que referencia a instância).

```
nomes = ['João', 'Paulo', 'Jorge', 'Ringo']
pessoas = [Pessoa(nome) for nome in nomes]
print(Pessoa.num_pessoas)
print(pessoas[0].nome)
```

Saída:

```
4
João
```

Exemplo: criamos a classe `Calculadora` com o método `soma`. Note que a variável `res`, declarada dentro do método `soma`, não aparece em nenhum dos atributos `__dict__`, nem no da instância, nem no da classe, pois a variável de método `res` só existe enquanto a função `soma` é executada:

```
class Calculadora:

    def soma(self, a, b):
        res = a + b
    return res


calculadora = Calculadora()
calculadora.soma(2, 3)
```

Saída:

```
5
```

6.4 Exemplos práticos

Vejamos alguns exemplos de código comentados a seguir, que ilustram os principais conceitos de Orientação a Objetos que já estudamos:

Exemplo de código 1: definição de classe e instanciação dos objetos

Definiremos a classe `Aluno` e criaremos dois objetos do tipo `Aluno`:

```
# Definição da classe
class Aluno:

    def __init__(self):
        self.nome = "Zé"
        self.idade = 18

    def dobra_idade(self):
        return self.idade * 2

# Instanciação dos objetos: criando (instanciando) 2 objetos do tipo (da classe) Aluno
```

```
# a variável aluno_1 guarda um objeto da classe Aluno, que contém nome,
idade e dobra_idade
aluno_1 = Aluno()
aluno_1.nome = "Maria" # definindo que o nome do aluno_1 é "Maria"

# a variável aluno_2 guarda OUTRO objeto da classe Aluno, que contém nome,
idade e dobra_idade
aluno_2 = Aluno()
aluno_2.idade = 21 # definindo que a idade do aluno_2 é 21

print(aluno_1.nome) # imprime Maria (valor que definimos)
print(aluno_2.nome) # imprime Zé (valor default)

print(aluno_1.dobra_idade()) # dobra a idade 18 (valor default)
print(aluno_2.dobra_idade()) # dobra a idade 21 (valor que definimos)

# ao mudar o valor de nome em aluno_1, o aluno_2 continua inalterado
aluno_1.nome = "Ana"
print("Nome do aluno_1:", aluno_1.nome)
print("Nome do aluno_2:", aluno_2.nome)
```

Saída:

```
Maria
Zé
36
42
Nome do aluno_1: Ana
Nome do aluno_2: Zé
```

Exemplo de código 2: parâmetro `self`

Como vimos, o parâmetro `self` é o primeiro parâmetro passado para qualquer método de classe, e o Python usará `self` para referenciar o objeto que está sendo criado:

```
class Calculadora:

    def __init__(self):
```

```
        self.valor = 5

    def add(self, numero):
      # não podemos chamar a variável "valor" sem "self" porque "valor" é
um atributo de Calculadora
      return self.valor + numero;

total = Calculadora()
print("total = %d" % total.add(3)) # self pegará o valor 5 e o valor 3 vai
para numero
```

Saída:

```
total = 8
```

Exemplo de código 3: método `__init__`

Vimos que o método `__init__` é usado para inicializar os objetos criados pela classe:

```
class Carro:

    def __init__(self):
        self.cor = "preto"

# um novo carro é criado com cor = "preto"
carro_1 = Carro()
print(carro_1.cor)

# alterando o valor da color de carro_1 para "branco"
carro_1.cor = "branco"
print(carro_1.cor)
```

Saída:

```
preto
branco
```

Você pode acessar o código-fonte desses exemplos no link https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_oo_python.ipynb.

6.5 Os 4 pilares da Orientação a Objetos

Encapsulamento

Encapsulamento é um recurso da Orientação a Objetos que permite agrupar código e os dados que esse código manipula em uma única entidade, protegendo estes elementos contra interferências externas, acesso indevido e/ou utilização/modificação inadequada por outro trecho de código definido fora do código encapsulado.

Uma classe naturalmente encapsula o código e os dados que a constituem, e é possível restringir o acesso a um membro (atributo ou método) da classe, permitindo acesso apenas para a classe em questão (*private*) ou em qualquer ponto do código (*public*).

Em OO, é uma boa prática tornar os atributos privados, uma vez que é responsabilidade de cada classe controlar os seus atributos. Algumas linguagens utilizam o conceito de palavras-chave para definir o nível de visibilidade, e Python considera que existem os tipos *public* e *non-public*. Para marcar como *non-public*, usa-se dois *underscores* (__) antes do nome do atributo ou método. Não se usa o termo *private* em Python porque nenhum atributo é de fato mantido como *private*.

Exemplo: como `numero` é *public*, ele pode ser acessado sem erros.

```
class Conta:

    def __init__(self, numero, saldo):
        self.numero = numero
        self.__saldo = saldo

conta1234 = Conta(1234, 750.84)
conta1234.numero
```

Saída:

```
1234
```

Já neste exemplo, quando tentamos acessar `__saldo`, o interpretador acusou que o atributo `__saldo` (*non-public*) não existe na classe `Conta`:

```
conta1234.__saldo
```

Saída:

```
AttributeError                            Traceback (most recent call
last)
<ipython-input-2-8a0b030f5094> in <module>
----> 1 conta1234.__saldo

AttributeError: 'Conta' object has no attribute '__saldo'
```

Em Python não existem atributos realmente privados (apenas um alerta de que você não deveria estar tentando acessá-lo), sendo possível acessar `__saldo` fazendo:

```
conta1234._Conta__saldo
```

Saída:

```
750.84
```

Apesar de ser possível acessar e até alterar o valor de um atributo *non-public* em Python, isso não é uma boa prática. Ainda assim, é possível utilizar a função `dir` para verificar que o atributo `_Conta__saldo` pertence ao objeto, mas o atributo `__saldo`, não:

```
dir(conta1234)
```

Saída:

```
['_Conta__saldo',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
```

```
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'numero']
```

Vamos agora tentar alterar diretamente o valor de `__saldo`:

```
conta1234.__saldo = 1000
```

Nenhuma mensagem de erro foi exibida! Será que foi possível alterar o valor do saldo diretamente? Vamos novamente acessar `__saldo`:

```
conta1234._Conta__saldo
```

Saída:

```
750.84
```

Repare que o valor do saldo da conta continua o mesmo de antes. Mas repare:

```
dir(conta1234)
```

Saída:

```
['_Conta__saldo',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__saldo',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'numero']
```

Agora, além de o atributo `_Conta__saldo` pertencer ao objeto, temos também o atributo `__saldo`. Isso ocorre porque Python é uma linguagem dinâmica: foi criado um novo atributo `__saldo` para a variável `conta1234`, inicializado em tempo de execução, e diferente do atributo de instância `__saldo`.

Vale ressaltar que as mesmas regras de acesso para atributos valem para os métodos. Como boa prática, os atributos de instância são *non-public* e a maioria dos métodos, *public*. Isso possibilita que a conversa entre objetos seja feita por troca de mensagens, ou seja, acessando seus métodos, em vez de um objeto manipular diretamente atributos que não sejam seus:

```
class Conta:

    def __init__(self, numero, saldo):
        self.__numero = numero
        self.__saldo = saldo

    def consulta_saldo(self):
        return self.__saldo

conta1234 = Conta(1234, 750.84)
conta1234.consulta_saldo()
```

Saída:

```
750.84
```

Herança

A herança permite que as características e comportamentos descritos para caracterizar uma classe "pai" (superclasse) possam ser herdados pela sua classe "filha" (subclasse). Com a herança, é possível concentrar as partes comuns na classe pai e as partes específicas nas classes filhas. A figura a seguir ilustra esse conceito.

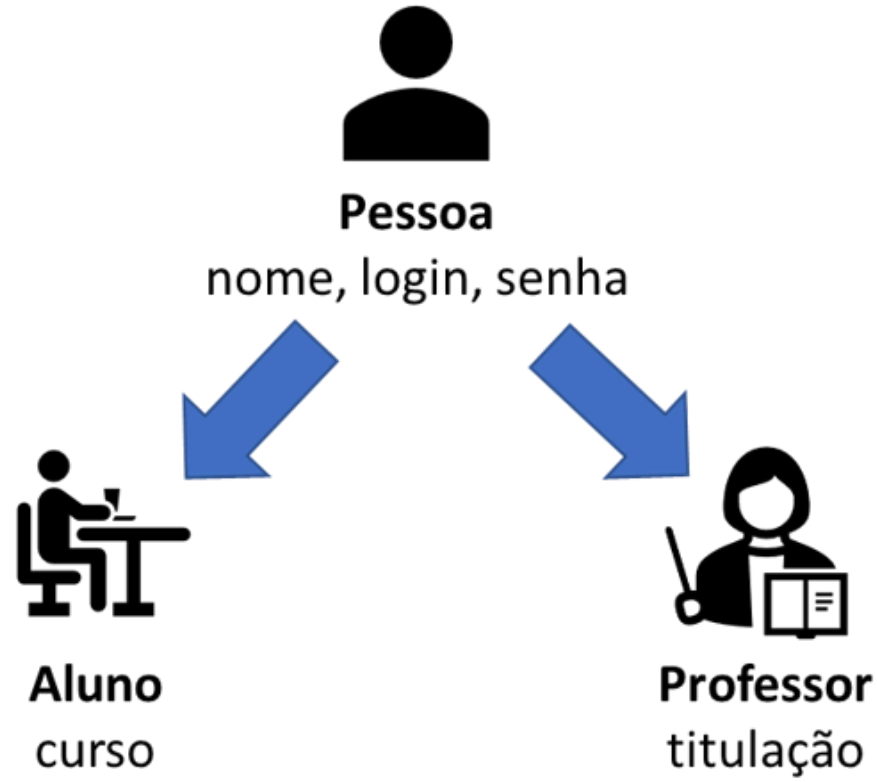


Figura 6.4: Exemplo de herança.

O trecho de código a seguir ilustra a definição da classe pai `Pessoa` e as classes filhas `Aluno` e `Professor`. Em seguida, são criados um objeto do tipo `Aluno` e um objeto do tipo `Professor`. Observação: Ao definir o método `__init__` nas classes `Aluno` e `Professor`, o método `__init__` da classe `Pessoa` não é herdado.

```
# Classe Pessoa
class Pessoa:

    def __init__(self, nome, login, senha):
        self.__nome = nome
        self.__login = login
        self.__senha = senha

    def consulta_nome(self):
        return self.__nome
```

```
# Classe Aluno
class Aluno(Pessoa):

    def __init__(self, nome, login, senha, curso):
        Pessoa.__init__(self, nome, login, senha)
        self.__curso = curso

    def consulta_curso(self):
        return self.__curso

# Classe Professor
class Professor(Pessoa):

    def __init__(self, nome, login, senha, titulacao):
        Pessoa.__init__(self, nome, login, senha)
        self.__titulacao = titulacao

    def consulta_titulacao(self):
        return self.__titulacao

pessoa1 = Pessoa('Maria', 'mary', 'm123')
print(pessoa1.consulta_nome())

aluna1 = Aluno('Viviane', 'vivi', 'v123', 'Informática')
print(aluna1.consulta_nome())
print(aluna1.consulta_curso())

prof1 = Professor('Tatiana', 'tati', 't123', 'Doutorado')
print(prof1.consulta_nome())
print(prof1.consulta_titulacao())
```

Saída:

```
Maria
Viviane
Informática
Tatiana
Doutorado
```

Polimorfismo

Segundo o exemplo anterior, podemos afirmar que todo `Aluno` é uma `Pessoa` (e todo `Professor` também), pois `Aluno` é uma extensão de `Pessoa` e herda suas propriedades e métodos. Assim, podemos nos referir a um `Aluno` (ou um `Professor`) como sendo uma `Pessoa`.

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas (o que não significa que o objeto pode se transformar em outro tipo). Se tivermos um método que espera receber um objeto do tipo `Pessoa`, ele pode receber no lugar um objeto do tipo `Aluno` ou do tipo `Professor`. Observe o exemplo a seguir:

```
class EntradaUniversidade:

    def __init__(self):
        pass

    def permite_entrada(self, pessoa):
        print("Pode entrar, " + pessoa.consulta_nome())

entrada = EntradaUniversidade()
entrada.permite_entrada(prof1)
```

Saída:

```
Pode entrar, Tatiana
```

```
# Classe Coordenador
class Coordenador(Pessoa):

    def __init__(self, nome, login, senha):
        Pessoa.__init__(self, nome, login, senha)

coord1 = Coordenador('Marcos', 'mk', 'm123')
entrada.permite_entrada(coord1)
```

Saída:

```
Pode entrar, Marcos
```

Repare que a funcionalidade representada pelo método `permite_entrada` foi construída antes mesmo que todos os tipos de `Pessoa` fossem criados, mas, ainda assim, ela continuou funcionando sem problemas após a criação da nova subclasse `Coordenador`, sem que fosse necessária nenhuma alteração no código.

O polimorfismo é um recurso que permite projetarmos nossos códigos em um alto nível de abstração (pensando em superclasses), sem nos preocuparmos com os níveis mais baixos (subclasses) que poderão, inclusive, ser criados posteriormente. Isso permite a criação de um código flexível, simples e de fácil manutenção. Usando o polimorfismo, é possível diminuir o acoplamento entre as classes, evitando que modificações no código resultem em modificações em diversos outros lugares.

Abstração

Ainda analisando o exemplo anterior, repare que não faz sentido termos no nosso sistema uma classe `Pessoa`, pois os usuários do sistema serão de tipos específicos como `Aluno`, `Professor`, `Coordenador`, mas nunca tão genéricos como `Pessoa`.

Entretanto, se removermos a classe `Pessoa` do nosso programa, perderemos duas importantes vantagens: o reúso de código entre as subclasses de `Pessoa` e a flexibilidade de termos um argumento polimórfico no método `permite_entrada`. A solução então é tornarmos `Pessoa` uma classe abstrata, o que significa que ela nunca poderá ser instanciada.

Em Python, uma classe abstrata deve conter pelo menos um método abstrato, e podemos criar uma classe abstrata herdando da superclasse para classes abstratas `ABC` (*Abstract Base Classes*), do módulo `abc`. Para isso, devemos importar esse módulo no nosso código. Observe como ficaria:

```
import abc
```

```
class Pessoa(abc.ABC):

    def __init__(self, nome, login, senha):
        self.nome = nome
        self.login = login
        self.senha = senha

    @abc.abstractmethod
    def consulta_nome(self):
        raise NotImplementedError()
```

Veja o que acontece se tentarmos instanciar agora um objeto do tipo `Pessoa`:

```
pessoa1 = Pessoa()
```

Saída:

```
TypeError                                 Traceback (most recent call
last)
<ipython-input-17-c8c545d518e4> in <module>
----> 1 pessoa1 = Pessoa()

TypeError: Can't instantiate abstract class Pessoa with abstract methods
consulta_nome
```

Observação: Não é possível instanciar uma subclasse de `Pessoa` sem implementar o método abstrato `consulta_nome`. Ao torná-lo abstrato, ele deverá ser obrigatoriamente implementado em todas as subclasses de `Pessoa`.

Associação, agregação e composição

Os conceitos de **associação**, **agregação** e **composição** permitem o relacionamento entre objetos. Ainda no exemplo anterior, imagine que queremos ampliar a classe `Aluno` para guardar nela qual é o seu professor orientador, criando nela um atributo `nome_orientador` para guardar o nome do professor orientador e outros atributos para guardar as demais informações do professor orientador, como a sua

titulação. Essa alternativa, além de não ser interessante (porque faria com que a classe `Aluno` começasse a ter atributos em excesso), não está de acordo com a programação OO.

Seguindo as boas práticas, podemos criar na classe `Aluno` um atributo `orientador`, do tipo `Professor`, que guardará todas as informações referentes ao orientador desse `Aluno`. Assim, os atributos de uma classe também podem ser referências para outras classes. Com isso, estamos fazendo uma **associação** de um `Professor` a um `Aluno`. Veja como ficaria no código:

```
class Aluno(Pessoa):

    def __init__(self, nome, login, senha, curso, orientador):
        Pessoa.__init__(self, nome, login, senha)
        self.__curso = curso
        self.__orientador = orientador

    def consulta_curso(self):
        return self.__curso

    def consulta_orientador(self):
        return self.__orientador

    def consulta_nome(self):
        return self.nome
```

Agora, ao criarmos um `Aluno`, precisaremos passar um `Professor` como orientador:

```
professorMarcos = Professor('Marcos', 'mk', 'm123', 'Doutor')

novoAluno = Aluno('Isabela', 'isa', 'i123', 'Engenharia', professorMarcos)
```

O professor orientador existe independente do aluno ao qual está associado, então dizemos que as classes `Professor` e `Aluno` estão relacionadas por uma **associação simples**. Quando semanticamente uma classe faz parte da outra, temos uma

agregação. Quando uma é membro da outra, temos uma **composição**.

Imagine agora que o aluno tem um histórico de ocorrências, com a data de matrícula e as ocorrências (notas obtidas, disciplinas cursadas, reprovações etc.). Neste caso, temos um exemplo de **composição**:

```
import datetime

class Historico():

    def __init__(self):
        self.__data_matricula = datetime.datetime.today()
        self.__ocorrencias = []

    def imprime(self):
        print("Matriculado em {}".format(self.__data_matricula))
        print("Ocorrências:")
        for o in self.__ocorrencias:
            print("- ", o)

    def add_ocorrencia(self, ocorrencia):
        self.__ocorrencias.append(ocorrencia)
```

Agora vamos modificar a classe `Aluno` para incluir um atributo do tipo `Historico`, e também vamos incluir dois novos métodos, `gera_ocorrencia` e `consulta_historico`:

```
class Aluno(Pessoa):

    def __init__(self, nome, login, senha, curso, orientador):
        Pessoa.__init__(self, nome, login, senha)
        self.__curso = curso
        self.__orientador = orientador
        self.__historico = Historico()

    def consulta_curso(self):
        return self.__curso
```

```
    def consulta_orientador(self):
        return self.__orientador

    def consulta_nome(self):
        return self.__nome

    def gera_ocorrencia(self, ocorrencia):
        self.__historico.add_ocorrencia(ocorrencia)

    def consulta_historico(self):
        self.__historico.imprime()
```

Vamos agora realizar a criação de um novo `Aluno`, a inclusão de duas ocorrências no seu histórico e a consulta do seu histórico:

```
novoAluno = Aluno('Isabela', 'isa', 'i123', 'Engenharia', professorMarcos)
novoAluno.gera_ocorrencia("Matriculou-se em Calculo 1")
novoAluno.gera_ocorrencia("Nota final de Calculo 1: 9,7")
novoAluno.consulta_historico()
```

Saída:

```
Matriculado em 2022-12-30 17:31:19.939527
Ocorrências:
-  Matriculou-se em Calculo 1
-  Nota final de Calculo 1: 9,7
```

Tanto a agregação quanto a composição são relacionamentos **todo-parte**. Na *agregação*, o objeto que compõe o todo tem uma parte do tipo de outro objeto e ambos podem existir separadamente (por exemplo, *Revista* e *Artigo*). Já na **composição**, se o objeto que representa o todo deixar de existir, o objeto que é uma parte relacionada a ele também deixará de existir (por exemplo, *Livro* e *Capítulo*).

Você pode acessar o código-fonte dos exemplos deste capítulo no link:

https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_oo_python.ipynb.

Capítulo 7
Boas práticas de projeto e construção de sistemas

Neste capítulo, vamos estudar assuntos que vão tornar nossos códigos mais profissionais. Primeiro, falaremos dos princípios SOLID, que são cinco princípios aplicados a Orientação a Objetos que visam facilitar a manutenção do código. Em seguida, vamos conhecer o guia de estilos Python, bem como o conceito de *Clean Code* e boas práticas de codificação em Python.

7.1 Princípios SOLID

Os princípios SOLID (MARTIN, 2017) buscam ajudar a realizar um bom projeto OO para um software de código flexível, escalável, sustentável e reutilizável. SOLID é um acrônimo dos cinco princípios:

1. ***Single Responsibility Principle* (SRP)** — Princípio de Responsabilidade Única;
2. ***Open/Closed Principle* (OCP)** — Princípio Aberto / Fechado
3. ***Liskov Substitution Principle* (LSP)** — Princípio da Substituição de Liskov
4. ***Interface Segregation Principle* (ISP)** — Princípio de Segregação de Interface
5. ***Dependency Inversion Principle* (DIP)** — Princípio da Inversão de Dependência

A seguir, apresentaremos cada um deles e veremos alguns exemplos envolvendo conceitos mais simples e didáticos, visando facilitar a compreensão dos princípios. Mais à frente neste livro, veremos exemplos da aplicação desses princípios no contexto de *Machine Learning*.

1. Princípio de Responsabilidade Única (SRP)

Esse princípio diz que uma classe não deve ter mais de um motivo para ser alterada, ou seja, uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software. Quando uma classe tem mais de uma responsabilidade e surge uma necessidade de alteração, será difícil modificar uma dessas responsabilidades sem comprometer as outras. Violar esse princípio pode trazer problemas como alto acoplamento, baixa coesão, dificuldades na implementação de testes automatizados e no reaproveitamento de código.

Vejamos um exemplo. A classe `Animal` possui mais de uma responsabilidade. Ela representa um animal e seus comportamentos e também é responsável por salvá-lo no banco de dados:

```
class Animal:

    def __init__(self, nome):
        self.__nome = nome

    def get_nome(self):
        return self.__nome

    def salvar(self):
        # Salva o animal no banco de dados
        pass
```

Aplicando o princípio SRP, teremos uma classe para cada responsabilidade:

```
class Animal:

    def __init__(self, nome):
        self.__nome = nome

    def get_nome(self):
        return self.__nome
```

```
class AnimalDAO:

    def salvar(self, animal: Animal):
        # Salva o animal no banco de dados
        pass
```

2. Princípio Aberto / Fechado (OCP)

Esse princípio diz que uma classe deve estar aberta para extensão, porém fechada para modificação. Quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código-fonte original.

Por exemplo, organizar o seu código com uma superclasse genérica e abstrata o suficiente permitirá que o programa seja estendido adicionando novas subclasses sem precisar alterar as classes existentes. À medida que tivermos novos animais no nosso programa, teremos que modificar a classe `Animal`. Aplicando o OCP, teremos uma classe para cada tipo de animal.

Sem o OCP:

```
class Animal:

    def __init__(self, nome):
        self.__nome = nome

    def get_nome(self):
        return self.__nome

    def faz_som(self):
        if self.__nome == "Cachorro":
            print("Au Au")
        if self.__nome == "Gato":
            print("Miau")
```

Com OCP:

```
class Animal:

    def __init__(self, nome):
        self.__nome = nome

    def get_nome(self):
        return self.__nome

    def faz_som(self):
        pass


class Cachorro(Animal):

    def faz_som(self):
        print("Au Au")


class Gato(Animal):

    def faz_som(self):
        print("Miau")
```

3. Princípio da Substituição de Liskov (LSP)

Esse princípio diz que uma subclasse deve poder ser substituída pela sua superclasse. Parece estar invertido, uma vez que sabemos que se *S* é uma subclasse de *C*, então os objetos do tipo *C* podem ser substituídos pelos objetos de tipo *S* em tempo de execução. Mas, quando o LSP é seguido, os métodos públicos definidos nas subclasses devem estar também presentes na superclasse, mesmo que de forma abstrata (o que também é conhecido como *strong behavioral subtyping*).

No fundo, o LSP encoraja o uso das classes de nível de abstração mais alto, estruturando as heranças do código de forma a ampliar as

possibilidades de aplicação do Princípio de Inversão de Dependência (DIP), uma vez que as abstrações (superclasses) terão os métodos necessários para expressar toda a semântica pretendida.

Vejamos um exemplo. O código a seguir não segue o LSP:

```
class Animal:
    pass


class Cachorro(Animal):

    def latir(self):
        print("Au Au")


class Gato(Animal):

    def miar(self):
        print("Miau")


class Dono:

    def passear(self, cachorro: Cachorro):
        cachorro.latir()
```

Aplicando o LSP:

```
class Animal:

    def faz_som(self):
        pass


class Cachorro(Animal):

    def faz_som(self):
        self.__latir()
```

```
    def __latir(self):
        print("Au Au")


class Gato(Animal):

    def faz_som(self):
        self.__miar()

    def __miar(self):
        print("Miau")


class Dono:

    def passear(self, animal: Animal):
        animal.faz_som()
```

4. Princípio da Segregação de Interfaces (ISP)

Esse princípio define que várias interfaces específicas são melhores do que uma interface genérica, ou seja, que uma classe não deve conhecer nem depender de métodos de que não necessite. Devemos criar interfaces segregadas com base nos requisitos, em vez de interfaces grandes de uso genérico.

No exemplo a seguir, podemos utilizar a classe `ImpressoraFazTudo` como base da classe `ImpressoraMultifuncional`, porque uma impressora multifuncional imprime, digitaliza e envia fax. Se utilizarmos a classe `ImpressoraFazTudo` como base para criar uma classe para uma `ImpressoraPadrao`, violaremos o princípio ISP, porque ela herdará métodos que não utilizará (`digitaliza` e `envia_fax`):

```
class ImpressoraFazTudo:

    def imprime(self):
        pass

    def digitaliza(self):
```

```
        pass

    def envia_fax(self):
        pass

class ImpressoraMultifuncional(ImpressoraFazTudo):

    def imprime(self):
        pass

    def escaneia(self):
        pass

    def envia_fax(self):
        pass
```

Utilizando o princípio ISP, vamos dividir a classe `ImpressoraFazTudo` em classes menores, para que as classes clientes implementem somente o que precisam:

```
class Impressora:

    def imprime(self):
        pass


class Digitalizadora:

    def digitaliza(self):
        pass

class Fax:

    def envia_fax(self):
        pass

class ImpressoraMultifuncional(Impressora, Digitalizadora, Fax):

    def imprime(self):
        pass
```

```
    def escaneia(self):
        pass

    def envia_fax(self):
        pass


# A classe ImpressoraPadrao pode utilizar somente o método que precisa:
class ImpressoraPadrao(Impressora):

    def imprime(self):
        pass
```

5. Princípio da Inversão de Dependências (DIP)

Esse princípio define que devemos depender de abstrações e não de implementações. Programe para uma interface (o supertipo de maior abstração), não para uma implementação (subtipo), e isso permitirá desacoplar classes clientes de implementações específicas. Vejamos um exemplo:

```
class Animal:

    def faz_som(self):
        pass


class Cachorro(Animal):

    def faz_som(self):
        self.latir()

    def latir(self):
        print("Au Au")


class Gato(Animal):
```

```
    def faz_som(self):
        self.miar()

    def miar(self):
        print("Miau")
```

Sem DIP:

```
class Dono:

    def passear(self, cachorro: Cachorro):
        cachorro.latir()
```

Com DIP:

```
class Dono:

  def passear(self, animal: Animal):
        animal.faz_som()
```

7.2 Guia de estilos

Com a finalidade de manter seu código organizado e fácil para que outros programadores (ou até você mesmo) possam compreender e dar manutenção, recomenda-se que uma boa programadora ou bom programador sigam boas práticas para escrever seus códigos. O criador do Python, Guido Von Rossum, é o principal autor do guia de estilos da linguagem Python.

A sigla **PEP** significa *Python Enhancement Proposal* e representa a forma com que a comunidade Python mantém as características e processos da linguagem atualizados. As PEPs podem ser utilizadas para envio, por qualquer pessoa, de propostas de alterações ou melhorias na linguagem. Através de um fluxo estabelecido, a comunidade discute, avalia criteriosamente cada proposta e decide implementá-la ou não nas próximas versões do Python.

Conforme descrito na **PEP-1**, as PEPs podem ser *Padrões* (descrevem novos recursos ou implementações de Python), *Informacionais* (problemas de design ou orientações gerais para a comunidade) e *Processos* (processos do Python), e todas as PEPs são indexadas pela **PEP-0** (https://peps.python.org/pep-0000/).

Além da PEP-0, algumas PEPs interessantes são:

- **PEP-8** (https://www.python.org/dev/peps/pep-0008/): guia de estilos que descreve a forma recomendada de se escrever um programa em Python. Seguir essa PEP para criar códigos em Python é muito importante, pois facilita a padronização e o entendimento entre desenvolvedores;
- **PEP-20** (https://www.python.org/dev/peps/pep-0020/): descreve o Zen do Python (*Zen of Python*), uma lista de curtos pensamentos que resumem como o Python funciona;
- **PEP-257** (https://www.python.org/dev/peps/pep-0257/): convenções de *docstrings* (comentários de módulos, funções, classes e métodos que se tornam a propriedade *doc* desses elementos e aparecem quando invocamos o *help* deles).

O seguinte código é um conhecido *Easter Egg* do Python:

```
import this
```

Ao executar esse código, o Zen do Python será exibido na saída:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

```
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Observação: apesar de guias de estilos para uma linguagem de programação serem uma excelente prática, pode ser difícil para programadores recordarem e aplicarem corretamente todas essas regras. Para dar apoio a essa tarefa repetitiva e passível de erros humanos, a comunidade Python criou programas que percorrem de forma automática os códigos-fonte, buscando alguma divergência dos estilos recomendados e que possam, inclusive, apresentar resultados indesejados ou padrões perigosos. Esses programas são conhecidos como ***linters***.

Layout de código

O primeiro pensamento do Zen do Python é *"Beautiful is better than ugly"*, ou seja, "Bonito é melhor que feio". Isso significa que a formatação de um código está diretamente relacionada ao seu entendimento. As orientações sobre layout de código da PEP-8 incluem:

- **Indentação**: usar 4 espaços (ou 1 `tab`) por nível de indentação, sendo os espaços a forma de indentação preferida. Porém, nunca devemos misturar tabulações e espaços;
- **Comprimento máximo de linhas**: as linhas de código do programa devem ter no máximo 79 caracteres. Para longos blocos de texto (*docstrings* ou comentários), devemos usar no máximo 79 caracteres. A melhor forma de continuar linhas longas é usando a continuação implícita (entre parênteses,

colchetes ou chaves), mas também é possível utilizar uma barra invertida, como neste exemplo extraído da PEP-8:

```
with open('/caminho/do/arquivo/que/sera/lido') as file_1, \
     open('/caminho/do/arquivo/que/sera/escrito') as file_2:
     file_2.write(file_1.read())
```

- **Linhas em branco**:
  - Separar uma definição de funções de uma definição da classe com duas linhas em branco;
  - Separar os métodos dentro de uma classe com uma única linha em branco, devendo haver também uma linha em branco entre a linha de definição da classe e seu primeiro método;
  - Usar linhas em branco para separar blocos lógicos dentro de métodos e funções.
- ***Imports***: devem ser sempre feitos em linhas separadas, nunca na mesma linha (mesmo que separados por vírgula). Por exemplo, este trecho de código é correto:

```
import pacote1
import pacote2
```

E este, incorreto:

```
import pacote1, pacote2
```

Os subpacotes ou funções de um mesmo pacote podem ser importados da seguinte forma:

```
from pacote import subpacote1, subpacote2
```

Os *imports* devem ser sempre colocados no início do arquivo de código, logo depois de quaisquer comentários ou *docstrings*, e antes de constantes ou globais, devendo ser utilizada uma linha em branco entre cada grupo, e devendo ser agrupados seguindo a ordem:

1. Módulos da biblioteca padrão;

2. Módulos terceiros relacionados entre si (por exemplo, todos os módulos de chamada de uma API proprietária usados na aplicação);
3. Aplicações locais/bibliotecas específicas da aplicação.

Nomenclatura

O segundo pensamento do Zen do Python é "*Explicit is better than implicit*", ou seja, "Explícito é melhor que implícito". Isso significa que a clareza quanto ao que queremos desenvolver em um bloco de código é muito importante, o que inclui nomear as variáveis, funções, classes e pacotes de forma clara e consistente.

As orientações sobre nomenclatura na PEP-8 são resumidas na tabela a seguir:

Tipo	Convenção de nomenclatura	Exemplos
Função	Use uma palavra ou palavras minúsculas. Palavras separadas por sublinhados.	*function, my_function*
Variável	Use uma letra, palavra ou palavras minúsculas. Palavras separadas com sublinhados.	*x, var, my_variable*
Classe	Comece cada palavra com uma letra maiúscula. Não separe palavras com sublinhados.	*Model, MyClass*

Tipo	Convenção de nomenclatura	Exemplos
Método	Use uma palavra ou palavras minúsculas. Palavras separadas com sublinhados.	*class_method, method*
Constante	Use uma letra, palavra ou palavras maiúsculas. Palavras separadas com sublinhados.	*CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT*
Módulo	Use uma palavra ou palavras curtas e minúsculas. Palavras separadas com sublinhados.	*module.py, my_module.py*
Pacote	Use uma palavra ou palavras curtas e minúsculas. Não separe palavras com sublinhados.	*package, mypackage*

Observação: evite utilizar letras únicas como *I* ou *O* para definir uma variável, pois pode ser facilmente confundida com 1 ou 0, por exemplo:

```
0 = 7 # isso pode parecer que estamos tentando atribuir 7 a zero
```

Comentários

Outro pensamento do Zen do Python relacionado ao guia de estilos é "*If the implementation is hard to explain, it's a bad idea*", ou seja, "Se a implementação é difícil de explicar, é uma má ideia". Isso

reforça a importância de a pessoa programadora sempre documentar de forma objetiva e precisa o seu código, o que facilita a sua manutenção e interpretabilidade. Algumas recomendações sobre comentários de código listadas na PEP-8 são:

- Linhas de comentários e *docstrings* devem ter no máximo 72 caracteres;
- Use frases completas para comentários, sempre começando com uma letra maiúscula;
- Atualize os comentários sempre que alterar o seu código;
- Use blocos de comentário para documentar as seções do seu código e ajudar outros programadores a compreenderem seu propósito e funcionalidade, observando as seguintes orientações:
  - Utilize nos comentários a mesma indentação do código que eles descrevem;
  - Inicie cada linha com um `#` seguido por um único espaço;
  - Separe os parágrafos por uma linha contendo um único `#`.

Vejamos um exemplo:

```
def calcular_equacao_quadratica(a, b, c, x):
    # Calcular a solução para uma equação quadrática
    #
    # Há sempre duas soluções para uma equação quadrática, x_1 e x_2
    x_1 = (-b+(b**2-4*a*c)**(1/2))/(2*a)
    x_2 = (-b-(b**2-4*a*c)**(1/2))/(2*a)
    return x_1, x_2
```

- Comentários *inline* são os escritos na mesma linha do seu respectivo código, e são úteis para explicar o que faz determinada linha de código, porém, devem ser usados com moderação e separados do código por pelo menos 2 espaços. Eles devem ser iniciados com um `#` seguido por um único espaço, assim como os comentários de blocos;
- Não use comentários para explicar o óbvio, como no exemplo a seguir:

```
idade = 7  # este é um comentário inline
nome = 'Tatiana Escovedo'  # nome do estudante
```

7.3 Clean code

O termo *clean code* (código limpo) se popularizou com o livro de Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, publicado em 2008. Esse livro reforça que um código ruim pode até mesmo funcionar, mas ele terá, provavelmente, uma legibilidade ruim, o que aumenta sua complexidade e custo de manutenção. Assim, as boas práticas de um código limpo consistem em instruções direcionadas ao modo como escrevemos nosso código e nas melhores práticas para realizar determinada tarefa.

A seguir, apresentaremos algumas boas práticas direcionadas para códigos em Python, que complementam ou reforçam as orientações do guia de estilos apresentadas anteriormente. Uma observação importante é que alguns dos trechos de código a seguir não funcionam isoladamente, pois são apenas trechos ilustrativos para exemplificar os conceitos aqui apresentados.

Nomes significativos

Algumas regras simples para a criação de bons nomes de variáveis, funções, parâmetros, classes, pacotes e arquivos-fonte incluem:

- Use nomes que revelem seu propósito. Se um nome requer um comentário, ele não revela seu propósito;
- Faça distinções significativas. O nome de uma variável jamais deve conter a palavra "variável", nem o nome de uma classe deve conter a palavra "classe";
- Use nomes passíveis de busca. Evite nomes de uma só letra, pois é difícil localizá-los ao longo de um texto;
- Não faça trocadilhos. Evite usar a mesma palavra para dois propósitos;

- Não use abreviações. Elas requerem um esforço maior na hora de ler o código, como ilustra o exemplo a seguir:

```
ct_n = len([1,2,3,4,5])  # Ruim
quantidade_de_numeros = len([1,2,3,4,5])  # Bom
nmdapsa = 'Josivaldo'  # Ruim
nome_da_pessoa = 'Josivaldo'  # Bom
```

- Utilize nomes descritivos. Se for necessário para que o código seja legível, utilize nomes grandes. Por exemplo:

```
def ltTParaMl(x): # Ruim
    return x * 1000

def transforma_litro_para_mililitro(valor_em_lt): # Bom
    return valor_em_lt * 1000
```

- Evite *números mágicos*: não utilize valores constantes soltos pelo código, sem identificação do que significam. Por exemplo:

```
# Ruim
if soma > 1250:  # O que significa este número?
    faca_algo()

# Bom
MINIMO_NECESSARIO_PARA_DESCONTO = 1250 # Bom
if soma > MINIMO_NECESSARIO_PARA_DESCONTO:
    faca_algo()

# Ruim
if variavel in (1,15,99,120):
    raise Exception('Não aceito')

# Bom
planos_nao_permitidos = (1,15,99,120)
if variavel in planos_nao_permitidos:
    raise Exception('Não aceito')
```

- Ao instanciar os objetos, é importante deixar claro o que cada um deles significa:

```
class Pessoa():
    def falar(self):
        print('Oi')

# Ruim
p1 = Pessoa()
p2 = Pessoa()

# Bom
pai = Pessoa()
filho = Pessoa()
```

Sempre pergunte o que você quer saber de forma clara

Não é uma boa prática fazer perguntas invertidas (por exemplo, verificar se um elemento não está contido em uma lista ou se um valor não é igual a outro). Veja um exemplo:

```
# Ruim
def lista_usuarios():
    if request not in ('POST','PUT','DELETE'):
        return get_users_list()
    else:
        return "Method not allowed"

# Bom
def lista_usuarios():
    if request.method == 'GET':
        return get_users_list()
    return "Method not allowed"
```

Não utilize booleanos negativos, mas, sim, operadores para invertê-los quando necessário. Por exemplo:

```
# Ruim
nao_deve_processar = True
if not nao_deve_processar:
    # faz algo aqui
    pass
# Bom
```

```
deve_processar = True
# ou
deve_processar = False
if deve_processar:
    # faz algo aqui
    pass
# ou
if not deve_processar:
    # faz algo aqui
    pass
```

Utilize booleanos de forma implícita

Os booleanos são mais claros e rápidos de entender quando utilizados como no exemplo a seguir:

```
# Ruim
def contem_letra(letra, nome):
    if letra in nome:
        return True
    else:
        return False

# Bom
def contem_letra(letra, nome):
    return letra in nome
# ou
# Ruim
if conta.numero in get_lista_de_contas_bloqueadas():
    conta.bloqueada = True
# Bom
conta.bloqueada = (conta.numero in get_lista_de_contas_bloqueadas())
```

Cuidado com a quantidade de parâmetros

Se precisamos receber muitos parâmetros em um método, devemos optar por receber estruturas de dados como dicionários (*dict*), listas (*list*), tuplas (*tuple*) e instâncias de alguma classe, em vez de uma listagem imensa de parâmetros. Por exemplo:

```
# Ruim
def metodo_com_muitos_parametros(
    nome, idade, data_nascimento, salario, funcao):
    # faz algo aqui...
    pass

# Bom
def metodo_com_muitos_parametros(dados_do_usuario):
    nome = dados_do_usuario.get('nome', None)
    # faz algo aqui...
    pass
```

Essa é uma boa prática pois, se for necessário adicionar ou remover parâmetros neste método, não será necessário mudar todas as suas chamadas.

Comentários são vilões quando não são bem utilizados

Evite comentários com o objetivo de explicar um código mal-escrito e difícil de entender. Neste exemplo, deveríamos reescrever o código:

```
# Ruim
# verifica se a venda entra na regra do desconto promocional
if venda.valor > 120 and venda.itens > 12:
    venda.desconto = 12
# Bom
if venda.permite_desconto_promocional():
    venda.aplica_desconto_promocional()
```

Também não devemos utilizar comentários óbvios, que só prejudicam a leitura, como neste exemplo:

```
def soma(a,b):
    # retorna a soma de a com b
    return a + b
```

Você pode acessar o código-fonte desses exemplos no link https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_boas_praticas.ipynb.

Alguns dos exemplos deste capítulo foram baseados no livro *Jornada Python: uma jornada imersiva na aplicabilidade de uma das mais poderosas linguagens de programação do mundo*, no qual a autora Tatiana Escovedo contribuiu com a curadoria.

Parte IV – Tópicos de Ciência de Dados

CAPÍTULO 8
Análise exploratória e visualização de dados

Lembrando do esquema básico de um projeto de Ciência de Dados, apresentado anteriormente neste livro, neste capítulo vamos focar na etapa de **coleta e análise de dados**, como ilustra a figura a seguir:

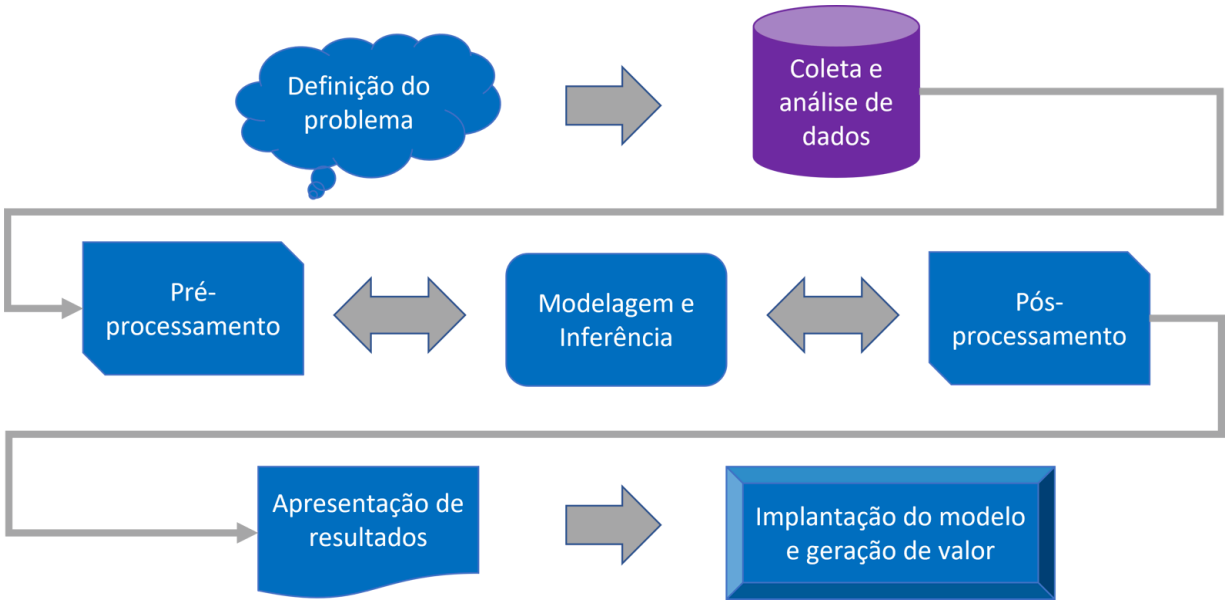


Figura 8.1: Esquema básico de um projeto de Ciência de Dados – Etapa de coleta e análise de dados.

Essa etapa inclui a **análise exploratória de dados**, e é uma etapa crítica porque é muito importante que você primeiro compreenda bem os seus dados para depois explorar as possíveis soluções. Para isso, além de utilizar conceitos da Estatística Descritiva, podemos

utilizar técnicas de *visualização de dados* (gráficos), o que ajuda a entender melhor seus dados, identificar valores discrepantes, faltantes ou inválidos, verificar a necessidade de alguma transformação de dados, identificar atributos redundantes, entre outros. Após entender e tratar seus dados, você poderá apresentá-los de forma mais adequada, bem como obter os melhores resultados possíveis nos algoritmos de *Machine Learning*.

8.1 Estatística Descritiva

A Estatística é um conjunto de técnicas que permite organizar, descrever, analisar e interpretar dados advindos de diversas origens a fim de extrair conclusões de forma sistemática. Pode ser subdivida em três grandes áreas:

- **Estatística Descritiva**: organiza e descreve dados, que podem ser expressos em tabelas e gráficos;
- **Probabilidade**: descreve os fenômenos aleatórios, ou seja, aqueles em que está presente a incerteza;
- **Inferência Estatística**: é o estudo de técnicas que possibilitam a um grande conjunto de dados a extrapolação das informações e conclusões obtidas a partir da amostra.

A Estatística Descritiva é muito utilizada na etapa de análise exploratória de dados de projetos de Ciência de Dados, tratando das **medidas de posição** (ou de tendência central), que permitem saber o grau de concentração dos dados, tais como *média*, *moda*, *mediana* e *medidas separatrizes*; e das **medidas de dispersão**, que permitem saber o grau de dispersão dos dados em torno de uma medida de posição (geralmente a média), tais como *desvio padrão*, *variância* e *coeficiente de variação*. A seguir, estudaremos alguns conceitos básicos da Estatística Descritiva.

Em muitos casos, precisamos de amostras da população para realizar uma pesquisa, uma vez que utilizar toda a população pode não ser possível ou adequado. Imagine que você precisa avaliar se está com anemia por meio de um exame de sangue, e em vez de uma *amostra* do seu sangue, o médico usa a população, ou seja, *todo* o seu sangue? Isso não é viável, por isso se faz necessária a retirada de uma amostra. Em Estatística, a **população** é a coleção de todos os indivíduos que possuem determinadas características, as quais estamos interessados em estudar, enquanto a **amostra** é um subconjunto da população, uma parte dos indivíduos que possuem determinadas características.

A Estatística Descritiva trabalha com dados, e chamamos de **dados brutos** os dados na sua forma mais primitiva, desprovidos de ordenação, assim que coletados, e de **rol estatístico** os dados brutos já ordenados, em ordem crescente ou decrescente. Para analisar dados, muitas vezes os organizamos em tabelas (dados tabulados) e podemos expressá-los em tabelas de frequência, podendo utilizar a *frequência absoluta simples*, a *frequência relativa simples*, a *frequência absoluta acumulada* ou a *frequência relativa acumulada*.

- A **frequência absoluta simples** é a contagem simples de elementos;
- A **frequência relativa simples** é a contagem simples de elementos, divididos pela soma das frequências simples, representando a proporção ou o percentual de observações;
- A **frequência absoluta acumulada** é a contagem acumulada até a classe de interesse (inclusive ela;)
- A **frequência relativa acumulada** é a contagem acumulada até a classe de interesse, dividida pela soma das frequências simples.

As figuras a seguir exemplificam esses conceitos:

Idade(Anos)	f_i
10	4
30	8
50	4
70	3
90	1

Idade(Anos)	fr
10	4/20 = 0,2 ou 20%
30	8/20 = 0,4 ou 40%
50	4/20 = 0,2 ou 20%
70	3/20 = 0,15 ou 15%
90	1/20 = 0,05 ou 5%

Figura 8.2: Frequência absoluta simples e frequência relativa simples.

Idade(Anos)	f_{ac}
10	4
30	(8 + f1) = 12
50	(4 + f1 + f2) = 16
70	(3 + f1 + f2 + f3) = 19
90	(1 + f1 + f2 + f3 + f4) = 20

Idade(Anos)	fr_{ac}
10	4/20 = 0,2 ou 20%
30	12/20 = 0,6 ou 60%
50	16/20 = 0,8 ou 80%
70	19/20 = 0,95 ou 95%
90	20/20 = 1 ou 100%

Figura 8.3: Frequência absoluta acumulada e frequência relativa acumulada.

Dentro da Estatística Descritiva, as **variáveis** são características associadas a uma população e podem ser classificadas em:

- **Qualitativas** — representam *atributos* ou *qualidades*, sendo:
  - **Ordinais** — quando existir uma ordem implícita, como classe social, grau de instrução e estágio da doença;
  - **Nominais** — quando não existir uma ordem implícita, tais como sexo, cor dos olhos, fumante/não fumante e doente/sadio.
- **Quantitativas** — representam *quantidades*, sendo:

- **Discretas** — quando for finita e enumerável, como número de filhos, número de carros e número de cigarros fumados por dia;
- **Contínuas** — quando os resultados possíveis pertencerem a um intervalo de números reais e resultados de mensuração, como peso, altura e salário.

A Estatística Descritiva pode ser dividida em dois grupos: *medidas de tendência central* (ou *de posição*) e *medidas de dispersão*, que detalharemos a seguir.

8.2 Medidas de tendência central

As **medidas de tendência central** (ou de posição) possibilitam saber o grau de concentração dos dados. São uma forma de resumir os seus dados por meio de valores representativos do conjunto de dados. São exemplos: *média*, *mediana*, *moda* e *medidas separatrizes*.

Média

A **média** é considerada uma medida volátil, uma vez que é afetada por valores extremos e sensível a valores atípicos (*outliers*). Ao se somar ou subtrair uma constante a todas as observações do conjunto, a média fica somada ou subtraída do valor da constante. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, a média fica multiplicada ou dividida pela mesma constante. Podemos utilizar diversos tipos de médias, conforme detalhado a seguir.

Tipos de média

1. Média aritmética (MA)

É a soma de todos os elementos do conjunto, dividida pelo número de elementos que compõem o conjunto. Sua fórmula é dada por:

$$\bar{x} = \frac{x_1 + x_2 + x_3 + \ldots + x_n}{n} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Figura 8.4: Média aritmética.

2. Média geométrica (MG)

É a raiz n-ésima do produto de todos os elementos que compõem o conjunto. Sua fórmula é dada por:

$$MG = \sqrt[n]{x_1 . x_2 . \ldots . x_n}$$

Figura 8.5: Média geométrica.

Observação: *n* é o número de elementos que compõem o conjunto.

3. Média harmônica (MH)

É o número de elementos, dividido pela soma dos inversos dos elementos que compõem o conjunto. Sua fórmula é dada por:

$$MH = \frac{n}{\left(\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}\right)}$$

Figura 8.6: Média harmônica.

4. Média ponderada (W)

Calculada por meio do somatório das multiplicações entre valores (x) e pesos divididos pelo somatório dos pesos (w). Sua fórmula é dada por:

$$W = \frac{\sum_{i=1}^{n} w_i X_i}{\sum_{i=1}^{n} w_i}$$

Figura 8.7: Média ponderada.

5. Média para dados agrupados em classes

Quando os dados estiverem agrupados em classes, usaremos a seguinte fórmula para média aritmética:

$$\bar{X} = \frac{\sum PM_i \cdot f_i}{\sum fi}, \text{ onde } PM_i = \frac{l_{inf} + l_{sup}}{2}$$

Figura 8.8: Média aritmética para dados agrupados em classes.

Apesar de a média ser fácil de computar, ela nem sempre é a melhor medida. Quando há uma alta variabilidade nos dados, medidas mais robustas como a *mediana* e a *moda* podem ser mais úteis, uma vez que, ao contrário da média, não são afetadas por valores extremos nem sensíveis a valores atípicos (*outliers*).

As médias geométrica e harmônica são menos conhecidas e pouco utilizadas por cientistas de dados, mas são bastante utilizadas pelos economistas ao calcular números índices para mensurar a inflação da economia de um país, deflacionar séries históricas de aluguéis e preço de imóveis, por meio de índices como IGP (Índice Geral de Preços), mantido pela FGV (Fundação Getúlio Vargas), e o IPCA (Índice de Preços ao Consumidor Amplo), calculado mensalmente

pelo IBGE (Instituto Brasileiro de Geografia e Estatística) para medir a variação dos preços no comércio.

Observação: Relação entre as médias aritmética, geométrica e harmônica: $MA \geq MG \geq MH$. O único caso em que $MA = MG = MH$ é quando todos os elementos possuem o mesmo valor no conjunto de dados.

Mediana

É o valor da variável que divide os dados ordenados em duas partes de igual frequência. A mediana é considerada uma medida robusta, uma vez que não é afetada por valores extremos. Ao se somar ou subtrair uma constante a todas as observações do conjunto, a mediana fica somada ou subtraída do valor da constante. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, a mediana fica multiplicada ou dividida pela mesma constante.

Para calcular a mediana em dados não agrupados em classes, deve-se utilizar o seguinte procedimento:

1. Colocar os dados em rol, ou seja, ordenar os dados de forma crescente ou decrescente;
2. Observar a paridade de n, pois o cálculo da mediana difere para n par e n ímpar;
3. Se n é ímpar, temos uma posição central única, dada por $PC\ (Posição\ Central) = (n+1)/2$. Após calcularmos a PC, a mediana será o valor que ocupa a posição central;
4. Se n é par, calcularemos duas posições centrais, $PC1 = n/2$ e $PC2 = (n/2) + 1$. Após calcularmos PC1 e PC2, a mediana será a média aritmética de PC1 e PC2.

Moda

É o valor que possui a maior frequência simples no conjunto de dados, consequentemente, o de maior probabilidade de ocorrência

em um conjunto de dados não agrupados em classes.

A moda é considerada uma medida robusta, uma vez que não é afetada por valores extremos. Ao se somar ou subtrair uma constante a todas as observações do conjunto, a moda fica somada ou subtraída do valor da constante. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, a moda fica multiplicada ou dividida pela mesma constante.

Para calcular a moda em dados não agrupados em classes, basta contar o número de ocorrências de cada valor. A moda será o valor com o maior número de ocorrências. Vale observar que, em um conjunto de dados, a moda pode ser única (*unimodal*), ter dois valores (*bimodal*) ou mesmo não existir (*amodal*).

Medidas separatrizes

As **medidas separatrizes** têm como objetivo dividir um conjunto de dados em *n* partes iguais. As mais utilizadas são os *quartis* e os *percentis*.

Os **quartis** dividem o conjunto em quatro partes iguais:

- Primeiro quartil (Q1): é o valor que deixa o conjunto de dados 25% abaixo dele e 75% para cima;
- Segundo quartil (Q2 = mediana): é o valor que deixa o conjunto de dados 50% abaixo dele e 50% para cima, em duas partes de igual frequência;
- Terceiro quartil (Q3): é o valor que deixa o conjunto de dados 75% abaixo dele e 25% para cima.

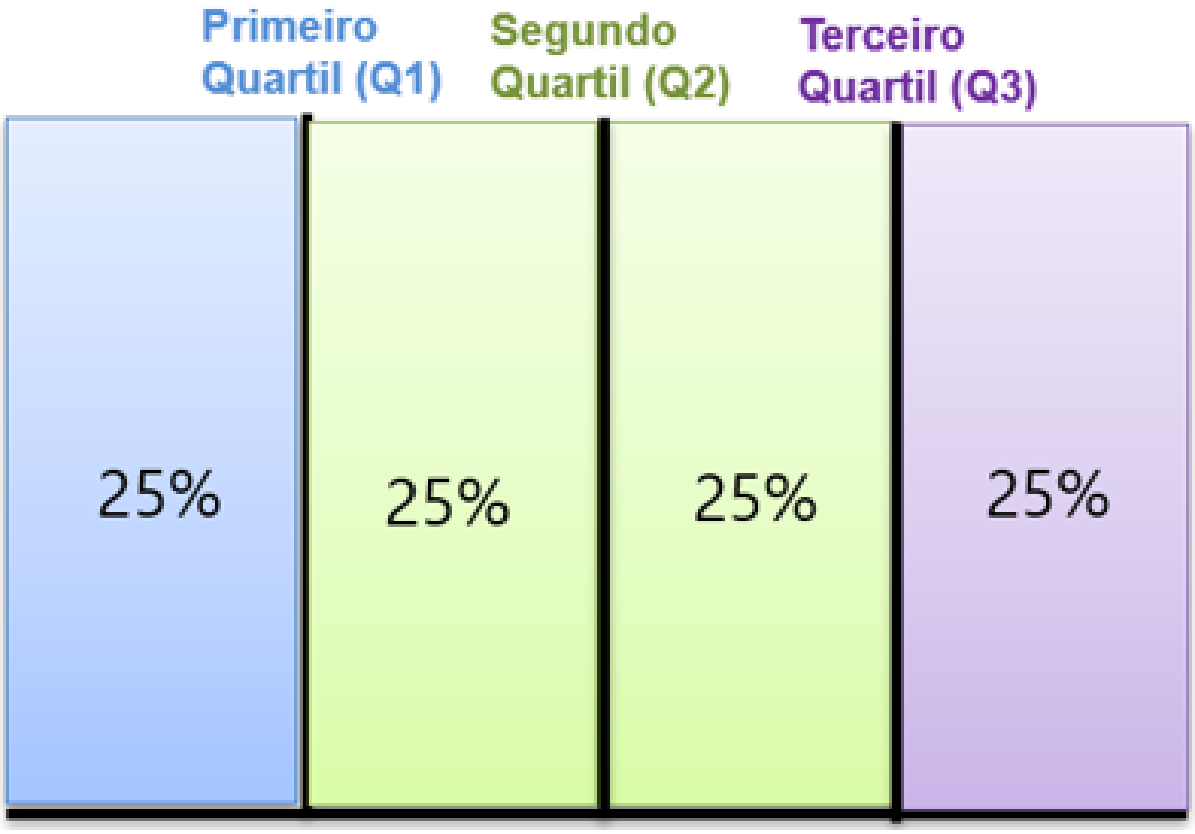


Figura 8.9: Quartis.

Os **percentis**: dividem o conjunto em 100 partes iguais.

- Percentil 25 (P25) = Q1;
- Percentil 50 (P50) = Q2 = Md;
- Percentil 75 (P75) = Q3.

Medidas de assimetria

As **medidas de assimetria** possibilitam analisar uma distribuição em relação a sua *moda*, *mediana* e *média*. Tentam mensurar como e quanto as distribuições se afastam da simetria da curva da distribuição normal de probabilidade.

Pense no conceito de simetria fazendo uma analogia a um espelho. Se traçarmos um eixo vertical no meio da curva dos dados e enxergarmos o mesmo de um lado e de outro, significa que seus

dados são simétricos. Nesse caso, a média, a mediana e a moda serão iguais, o que não será verdade no caso de dados assimétricos. Esse conceito é ilustrado pela figura a seguir:

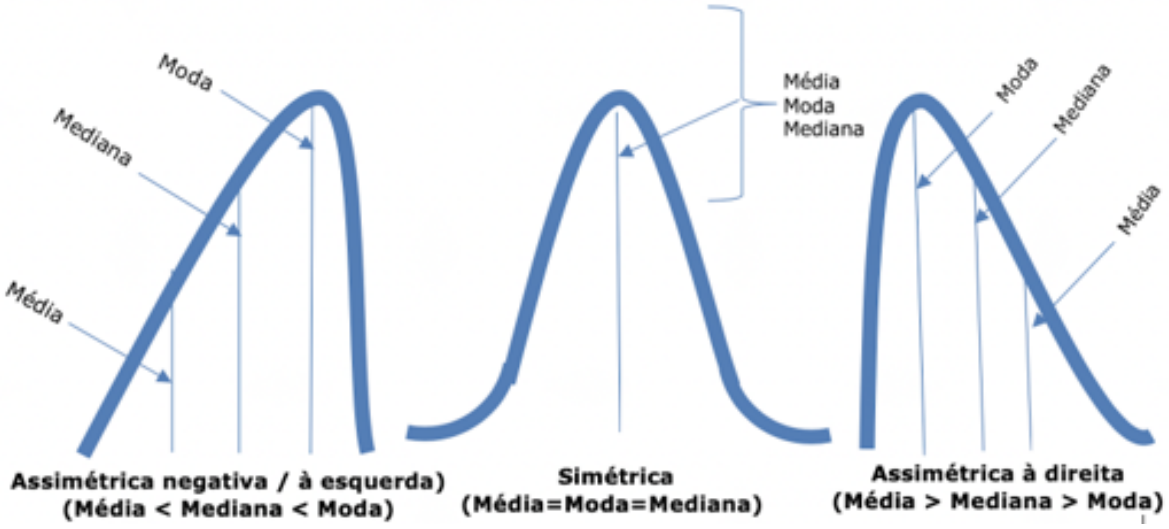


Figura 8.10: Medidas de assimetria.

Coeficiente de Assimetria de Pearson

Existem alguns coeficientes de assimetria de Pearson, mas falaremos somente do coeficiente de assimetria de Pearson baseado nos momentos, que tem a seguinte fórmula:

$$AS = \frac{\sum_i^N (X_i - \bar{X})^3}{(N-1) \times \sigma^3}$$

Figura 8.11: Coeficiente de assimetria de Pearson baseado nos momentos.

onde:

- AS = assimetria
- N = número de variáveis na distribuição
- X_i = variável aleatória

- $\bar{X}$ = média da distribuição
- σ = desvio padrão da distribuição

Quando *AS = 0*, a distribuição será simétrica, ou seja, terá ausência de assimetria. Ela terá o formato de uma distribuição normal de probabilidade. Quando *AS > 0*, a distribuição será assimétrica à direita ou positiva, ou seja, terá concentração dos valores no gráfico à esquerda e a curva se alongará mais à direita do gráfico. Quando *AS < 0*, a distribuição será assimétrica à esquerda ou negativa, ou seja, terá concentração dos valores no gráfico à direita e a curva se alongará mais à esquerda do gráfico.

Medidas de dispersão

As **medidas de dispersão**, também chamadas de **medidas de variação**, nos permitem saber o grau de dispersão dos dados em relação a uma medida de tendência central (geralmente a média). Elas medem se os dados estão compactados ou espalhados. São exemplos de medidas de dispersão: *amplitude*, *variância*, *desvio padrão*, *coeficiente de variação*. Vamos entender cada conceito associado a *população* e *amostra* e, em seguida, veremos um exemplo.

População

Amplitude populacional: é a diferença entre o maior e o menor valor da população: *H = (máximo - mínimo)*.

Variância populacional: é o valor médio dos quadrados dos desvios de cada valor da população em relação ao valor médio da população:

$$\sigma^2 = \frac{(x_1-\bar{x})^2 + (x_2-\bar{x})^2 + \cdots + (x_N-\bar{x})^2}{N} = \frac{\sum_{i=1}^{N}(x_i-\bar{x})^2}{N}$$

A *variância* é considerada uma medida de dispersão absoluta. Ao se somar ou subtrair uma constante a todas as observações do

conjunto, a variância fica inalterada. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, a variância fica multiplicada ou dividida pela mesma constante ao quadrado.

Desvio padrão populacional: é a raiz quadrada da variância populacional:

$$\sigma = \sqrt{\sigma^2}$$

O *desvio padrão* é considerado uma medida de dispersão absoluta. Ao se somar ou subtrair uma constante a todas as observações do conjunto, o desvio padrão fica inalterado. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, o desvio padrão fica multiplicado ou dividido pela mesma constante. É mais fácil de interpretar do que a variância por estar na mesma escala que os dados originais.

Coeficiente de variação populacional: é a razão entre o *desvio padrão populacional* e a *média populacional*, e é um indicador de homogeneidade dos dados da população:

$$CV_x = \frac{\sigma}{\mu},$$

O *coeficiente de variação* é considerado uma medida de dispersão relativa e é a melhor de todas as medidas de dispersão, uma vez que leva em consideração não somente a variabilidade como também a média do conjunto de dados. Ao se somar ou subtrair uma constante a todas as observações do conjunto, o denominador do coeficiente de variação fica somado ou subtraído do valor da constante. Ao se efetuar a multiplicação ou divisão por uma constante a todas as observações, o coeficiente de variação fica inalterado.

Observação: o coeficiente de variação é a única medida de variação adimensional (não possui unidade de medida). Em geral, consideramos um coeficiente de variação < 25% um bom indicador de homogeneidade dos dados.

Amostra

Amplitude amostral: é a diferença entre o maior e o menor valor da amostra: *h = (máximo – mínimo).*

Variância amostral: é o valor médio dos quadrados dos desvios de cada valor da amostra em relação ao valor médio da amostra:

$$s^2 = \frac{(x_1-\bar{x})^2 + (x_2-\bar{x})^2 + \cdots + (x_n-\bar{x})^2}{n-1} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}$$

Note que dividimos a variância amostral por *n-1*, diferentemente da variância populacional, que dividimos por *N*.

Desvio padrão amostral: é a raiz quadrada da variância amostral:

$$s = \sqrt{s^2}$$

Coeficiente de variação amostral: é a razão entre o desvio padrão amostral e a média amostral, e é um indicador de homogeneidade dos dados da amostra:

$$CV_x = \frac{s}{\bar{x}}$$

8.3 Gráficos para visualização de dados

Quando queremos observar o comportamento de determinadas variáveis de forma rápida e sintetizada, especialmente quando possuem um número elevado de observações, pode ser uma excelente ideia utilizar **gráficos**. Contudo, existem gráficos apropriados para cada tipo de variável. Para variáveis *qualitativas*, podemos destacar:

- **Gráfico de pizza**: apresenta a composição, usualmente em porcentagem, das partes de um todo. Consiste em um raio arbitrário, representando o todo, dividido em setores (fatias da pizza), que correspondem às partes de maneira proporcional;
- **Gráfico de barras/colunas**: retângulos ou barras, em que uma das dimensões é proporcional à magnitude a ser representada, sendo a outra arbitrária, contudo, igual para todas as barras. Essas barras são dispostas paralelamente às outras, horizontal ou verticalmente.

Esses gráficos são ilustrados pelas figuras a seguir:

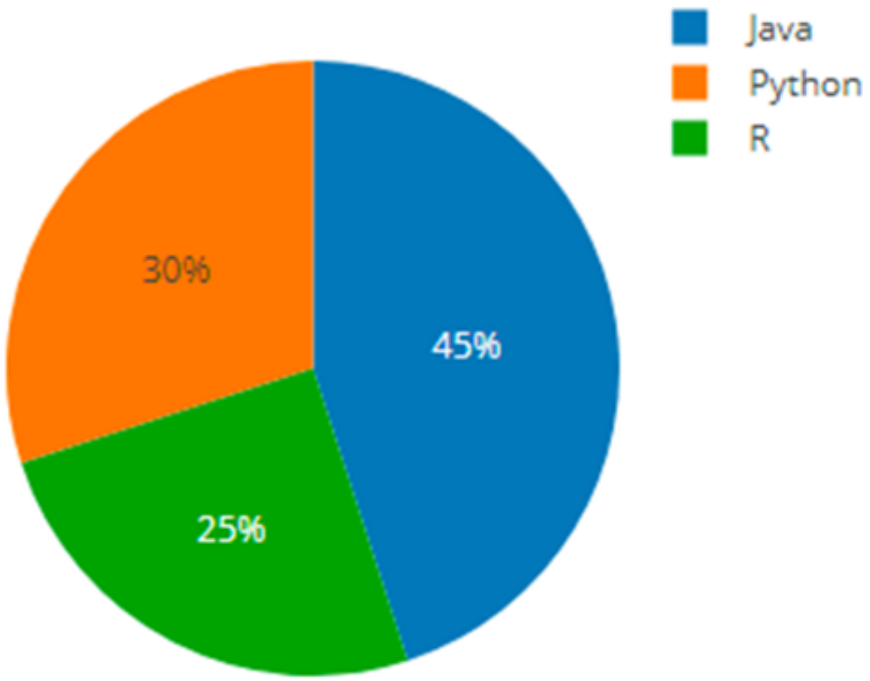


Figura 8.18: Gráfico de pizza.

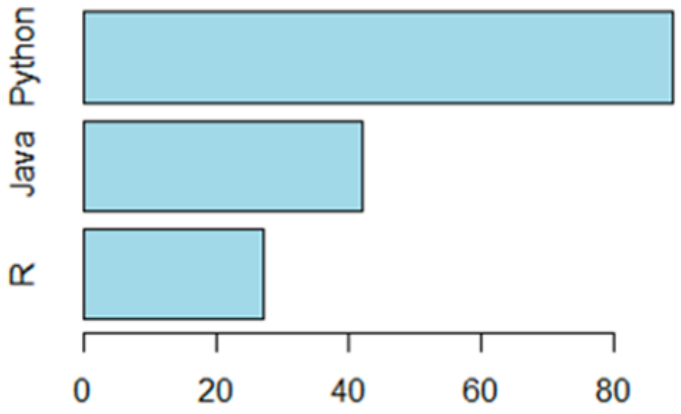


Figura 8.19: Gráfico de barras.

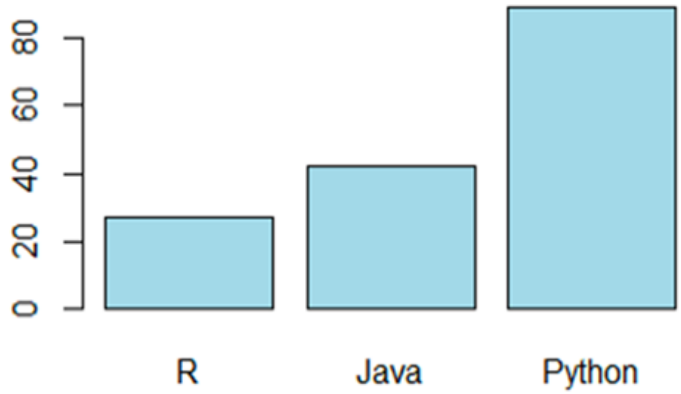


Figura 8.20: Gráfico de colunas.

Para as variáveis *quantitativas*, teremos uma maior variedade de gráficos. No caso de *variáveis quantitativas discretas*, geralmente organizamos os dados em **tabelas de frequência**, e a representação gráfica é mediante gráfico de barras, similar ao visto para as variáveis qualitativas. Elas também podem ser representadas por um **gráfico de dispersão (*scatter plot*)**, um gráfico similar ao gráfico de barras, porém são plotadas somente as interseções dos pontos de *x* e *y*, como ilustram as figuras a seguir:

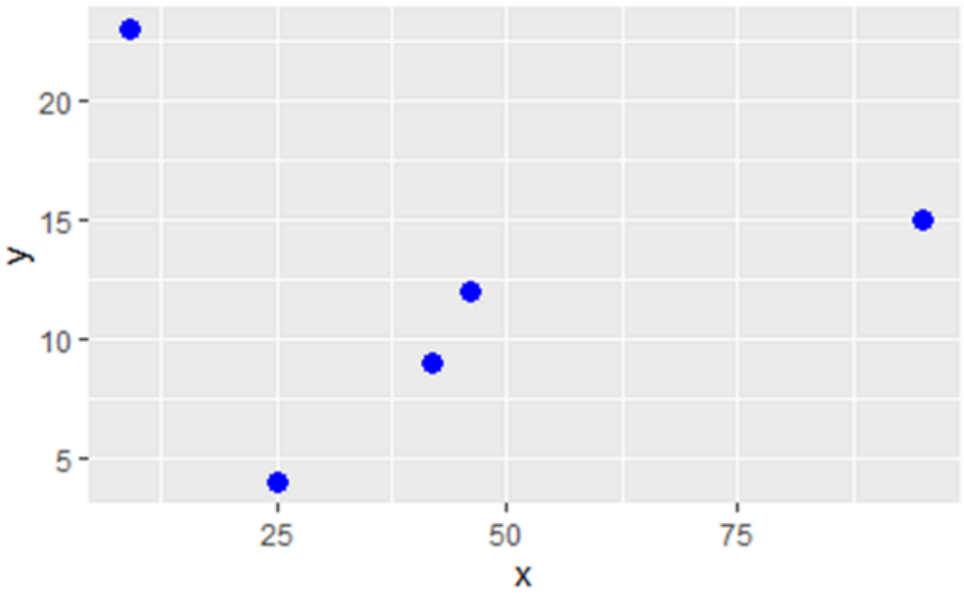


Figura 8.21: Gráfico de dispersão com poucos dados.

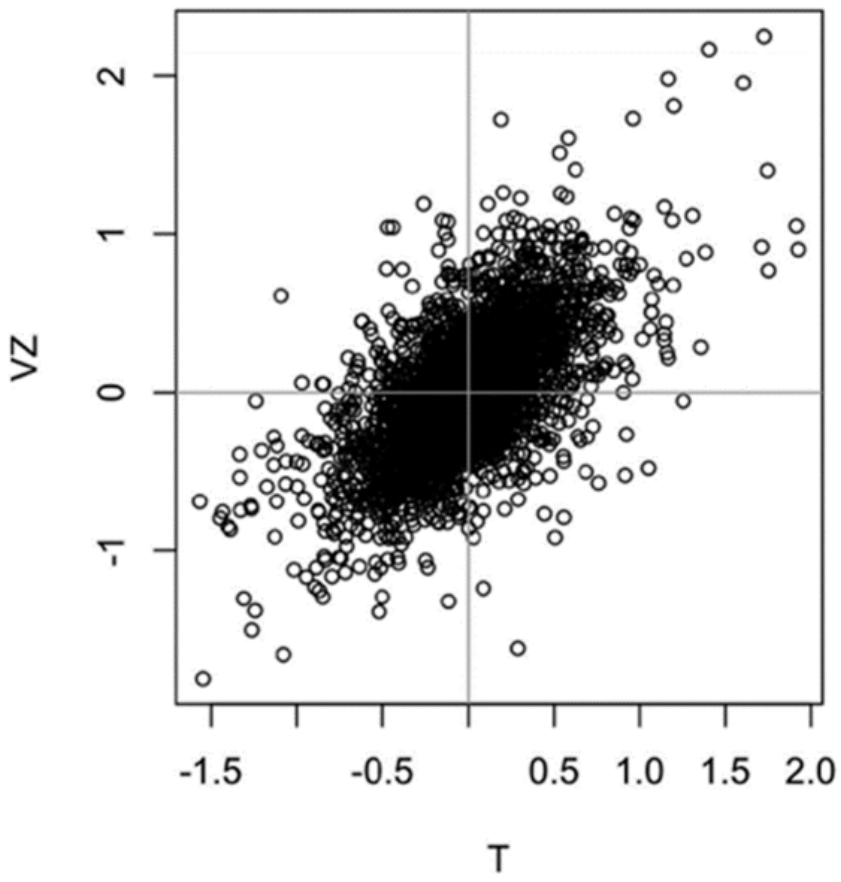


Figura 8.22: Gráfico de dispersão. Fonte: Bruce e Bruce, 2019.

Já para as *variáveis quantitativas contínuas*, podemos usar o **histograma** — um gráfico de barras contíguas, com as bases proporcionais aos intervalos de classe e a área do retângulo proporcional à respectiva frequência — ou o **gráfico de linhas** — utilizado quando estamos interessados em expressar variáveis que representam passagem de tempo, ou seja, em mostrar a evolução histórica. Esses gráficos são ilustrados pelas figuras a seguir:

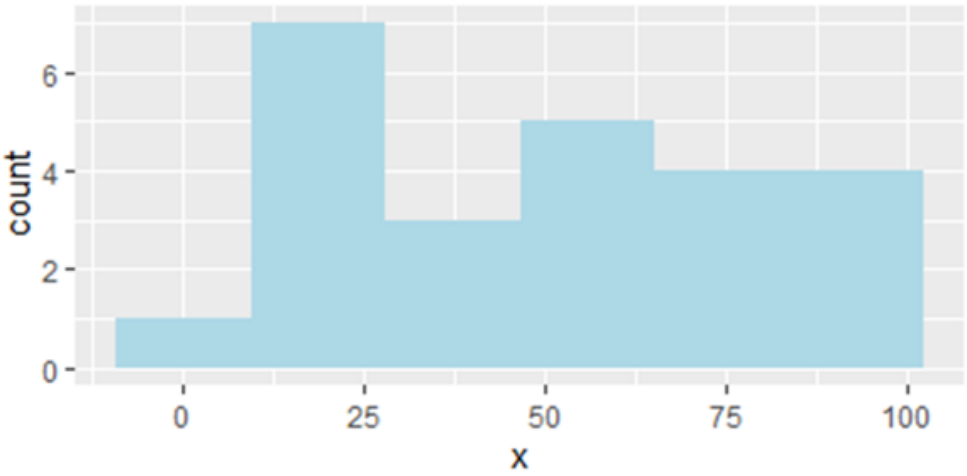


Figura 8.23: Histograma.

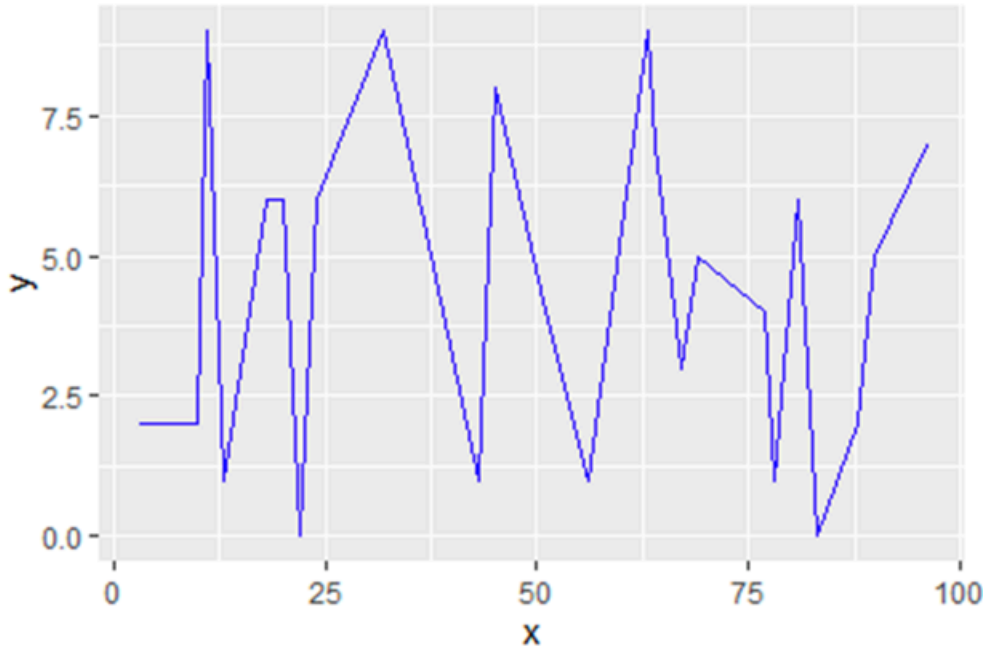


Figura 8.24: Gráfico de linhas.

Para trabalharmos com as medidas de tendência central e de dispersão graficamente, podemos usar o **diagrama de caixas (*boxplot*)**, que é uma representação gráfica da distribuição dos dados. Esse diagrama nos dá informação da assimetria da distribuição, da presença de *outliers* (valores atípicos) e da variabilidade dos dados por meio da amplitude (*Máx-Min*). No

boxplot, dentro do quadrado (*box*), encontramos 50% dos dados, delimitados pelo primeiro e terceiro quartis e pela mediana. Fora do quadrado, os delimitadores representados por pequenas linhas retas mostram o limite razoável dos dados, e os pequenos círculos fora desses delimitadores representam possíveis *outliers*. A figura a seguir ilustra o *boxplot*:

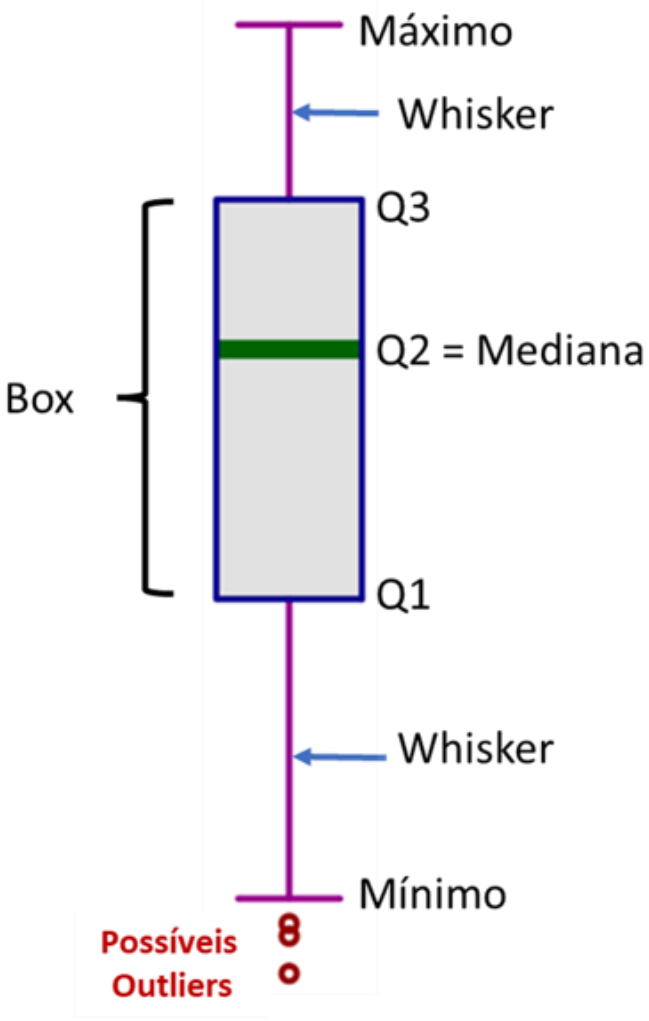


Figura 8.25: Boxplot.

Os *boxplots* também podem ser utilizados para comparar visualmente as distribuições de uma variável numérica agrupada conforme uma variável categórica, como ilustra a figura a seguir:

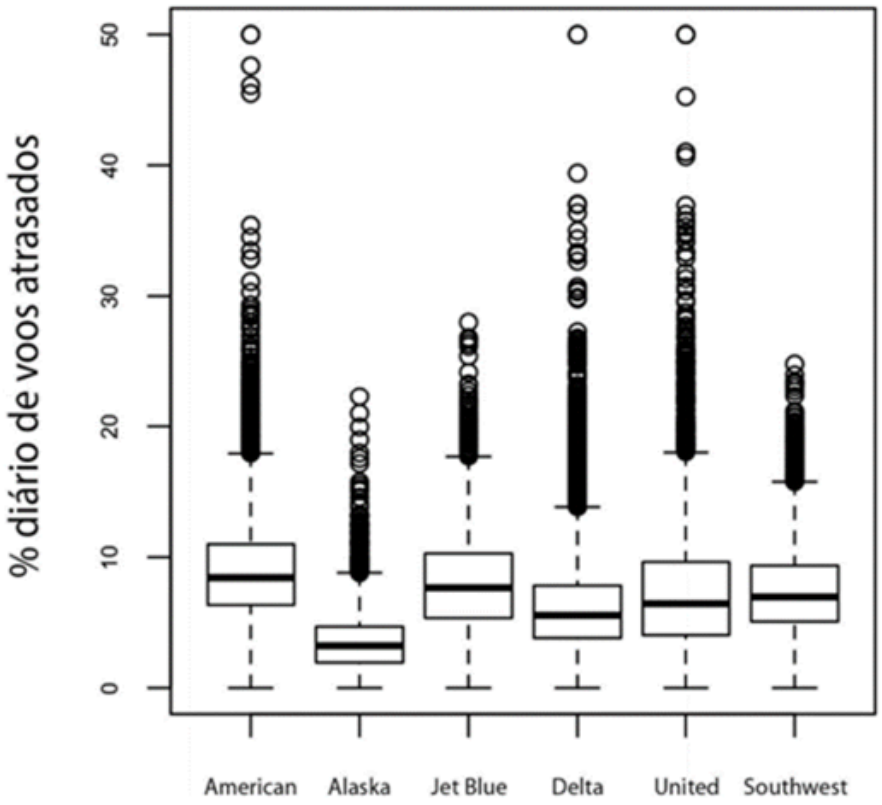


Figura 8.26: Boxplots comparativos. Fonte: Bruce e Bruce, 2019.

A análise exploratória de dados muitas vezes envolve o estudo da **correlação** entre preditores, e entre preditores e uma variável-alvo. Dizemos que as variáveis *x* e *y* têm correlação positiva se *x* aumenta quando *y* aumenta, e correlação negativa se quando *x* aumenta, *y* diminui (e vice-versa). O **coeficiente de correlação** mede o nível em que as variáveis numéricas estão associadas umas às outras, e varia de *-1* a *+1*, onde *0* significa ausência de correlação. Vale a pena ressaltar que o coeficiente de correlação é sensível a *outliers* nos dados.

A **matriz de correlação** é uma tabela na qual as variáveis são mostradas tanto nas linhas quanto nas colunas, e os valores das células são a correlação entre as variáveis, e o **gráfico de correlação** é a sua representação gráfica, como ilustra a figura a seguir:

	T	CTL	FTR	VZ	LVLT
T	1.000	0.475	0.328	0.678	0.279
CTL	0.475	1.000	0.420	0.417	0.287
FTR	0.328	0.420	1.000	0.287	0.260
VZ	0.678	0.417	0.287	1.000	0.242
LVLT	0.279	0.287	0.260	0.242	1.000

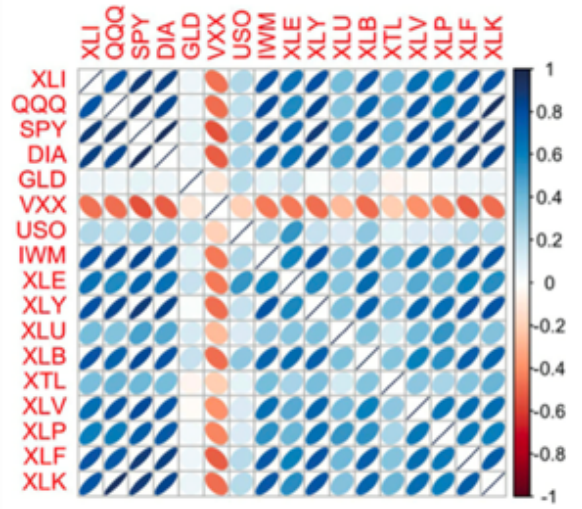


Figura 8.27: Matriz e gráfico de correlação. Fonte: Bruce e Bruce, 2019.

É muito importante sempre considerarmos a visualização adequada para o problema em questão. É possível encontrar mais informações sobre erros frequentes na representação visual de dados em Rodrigues *et al.* (2021).

8.4 Bibliotecas Python para análise exploratória

O **NumPy** (https://numpy.org/) é um pacote fundamental para a computação científica com Python. Essa biblioteca fornece diversas funções matemáticas e um *array* multidimensional de alto desempenho que é utilizado como base para diversas operações, principalmente em manipulação de dados, e suas estruturas de dados básicas são *arrays* e matrizes.

O **Pandas** (https://pandas.pydata.org/) é um pacote construído sobre o *NumPy* muito utilizado para manipular e analisar dados estruturados ou tabulares. Suas estruturas de dados principais são os *Dataframes* (matrizes multidimensionais similares a uma tabela de dados relacional, com linhas e colunas rotuladas) e *Series* (uma

única coluna, sendo que *Dataframes* possuem um ou mais *Series*). Essa biblioteca oferece uma funcionalidade de indexação que permite o *slice and dice* dos dados, agregações, seleção de *subsets* de dados, entre outros recursos para análise de dados.

O **Matplotlib** (https://matplotlib.org/) é um pacote que fornece uma API que é utilizada por diversas outras bibliotecas de plotagem gráfica. É uma excelente biblioteca para criar gráficos básicos como gráficos de linhas, de barras, histogramas e muitos outros. Esse pacote apresenta um baixo nível de implementação e, portanto, possibilita um alto nível de customização dos seus gráficos.

O **Seaborn** (https://seaborn.pydata.org/) é um pacote de visualização baseado em *Matplotlib* que oferece uma interface de alto nível para a criação de gráficos atrativos, profissionais e com análises estatísticas. Com essa biblioteca, é possível escrever em uma linha um gráfico que precisaria de diversas linhas usando apenas *Matplotlib*. Os métodos de visualização da biblioteca *Seaborn* têm uma sintaxe mais simples, porém são mais limitados, sendo necessário utilizar a biblioteca *Matplotlib* para customizações adicionais.

8.5 Exemplo prático

Você pode acessar o código-fonte do exemplo prático de análise exploratória de dados em Python pelo seguinte link: https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_analise_exploratoria.ipynb

Neste link você também consegue visualizar as imagens coloridas, caso esteja acessando uma versão em preto e branco deste livro.

Trabalhando com Pandas

Para exemplificar como funciona a etapa de análise exploratória de dados em um *dataset*, vamos utilizar o pacote Pandas, focando em algumas possibilidades interessantes de análises, mas sem a pretensão de ser um tutorial exaustivo desse pacote, que é muito amplo e rico. Para conhecer mais sobre o Pandas, você pode explorar um dos diversos tutoriais oficiais, como o disponível em https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html.

Iniciaremos esta prática importando o Pandas:

```
# importação do Pandas
import pandas as pd
```

Agora vamos importar o *dataset* que vamos utilizar a partir de um link. Armazenaremos os dados em um *dataframe*. Nesta prática, trabalharemos com o *dataset Iris*, um dos mais conhecidos e utilizados em *Machine Learning*, criado em 1936. Esse é um *dataset* pequeno e simples, ideal para uma primeira prática, e contém 150 instâncias (linhas), 4 variáveis numéricas (largura e comprimento da sépala e da pétala em centímetros) e 1 variável categórica (espécie da flor, podendo ser *setosa*, *versicolor* ou *virginica*).

```
# importando dados de uma url para um dataframe

# url a importar
url_dados =
'https://raw.githubusercontent.com/profkalinowski/livroescd/main/iris.data
'


# labels dos atributos do dataset
atributos = ['comprimento_sepala', 'largura_sepala', 'comprimento_petala',
'largura_petala', 'especie']

# carga do dataset através da url - há diversos parâmetros no read_csv que
podem ser interessantes, como sep, usecols e header
iris = pd.read_csv(url_dados, names=atributos)
```

Inicialmente, vamos utilizar alguns comandos da biblioteca Pandas para exibir algumas estatísticas descritivas e fazer uma rápida análise exploratória do *dataset*. Os comentários nos blocos de código auxiliam no seu entendimento.

```
# verificando que iris é um dataframe
type(iris)
```

Saída:

```
pandas.core.frame.DataFrame
```

```
# verificando os tipos de cada coluna do dataframe
iris.dtypes
```

Saída:

```
comprimento_sepala    float64
largura_sepala        float64
comprimento_petala    float64
largura_petala        float64
especie                object
dtype: object
```

```
# exibindo as primeiras linhas
iris.head()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Figura 8.28: Saída.

```
# exibindo as últimas linhas
iris.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figura 8.29: Saída.

```
# exibindo as primeiras 10 linhas
iris.head(10)
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

Figura 8.30: Saída.

```
# exibindo os nomes das colunas
iris.columns
```

Saída:

```
Index(['comprimento_sepala', 'largura_sepala', 'comprimento_petala',
       'largura_petala', 'especie'],
      dtype='object')
```

```
# exibindo as dimensões do dataset
iris.shape
```

Saída:

```
(150, 5)
```

```
# exibindo a quantidade de linhas por coluna
iris.count()
```

Saída:

```
comprimento_sepala    150
largura_sepala        150
comprimento_petala    150
largura_petala        150
especie               150
dtype: int64
```

```
# exibindo um sumário estatístico
iris.describe()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Figura 8.31: Saída.

```
# exibindo informações sobre colunas e uso de memória
iris.info()
```

Saída:

```
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   comprimento_sepala  150 non-null    float64
 1   largura_sepala      150 non-null    float64
 2   comprimento_petala  150 non-null    float64
 3   largura_petala      150 non-null    float64
 4   especie             150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

A fim de analisarmos os nossos dados de diversos ângulos, o Pandas permite a reorganização de um *dataframe*, por exemplo, ordenando-o por alguma coluna. Também podemos selecionar uma ou mais colunas e/ou linhas específicas, como mostram os exemplos a seguir. Os comentários nos blocos de código auxiliam no seu entendimento.

```
# ordenando o dataframe por uma coluna
iris.sort_values(by='largura_sepala')

# obs.: este comando retorna apenas o resultado da query. Se quisermos
alterar a ordenação do dataframe em si,
# teríamos que fazer: iris = iris.sort_values(by='largura_sepala')
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
60	5.0	2.0	3.5	1.0	Iris-versicolor
62	6.0	2.2	4.0	1.0	Iris-versicolor
119	6.0	2.2	5.0	1.5	Iris-virginica
68	6.2	2.2	4.5	1.5	Iris-versicolor
41	4.5	2.3	1.3	0.3	Iris-setosa
...	...	...	...	...	...
16	5.4	3.9	1.3	0.4	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa

150 rows × 5 columns

Figura 8.32: Saída.

```
# ordenando por uma coluna de ordem descendente
iris.sort_values(by='largura_sepala', ascending=False)
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
15	5.7	4.4	1.5	0.4	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
...	...	...	...	...	...
87	6.3	2.3	4.4	1.3	Iris-versicolor
62	6.0	2.2	4.0	1.0	Iris-versicolor
68	6.2	2.2	4.5	1.5	Iris-versicolor
119	6.0	2.2	5.0	1.5	Iris-virginica
60	5.0	2.0	3.5	1.0	Iris-versicolor

150 rows × 5 columns

Figura 8.33: Saída.

```
# selecionando uma coluna específica
iris['largura_sepala']
```

Saída:

```
0      3.5
1      3.0
2      3.2
3      3.1
4      3.6
      ...
145    3.0
146    2.5
147    3.0
148    3.4
149    3.0
Name: largura_sepala, Length: 150, dtype: float64
```

```
# selecionando um subconjunto de linhas consecutivas
iris[7:11]
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa

Figura 8.34: Saída.

```
# selecionando um subconjunto de colunas para todas as linhas
iris.loc[:, ['largura_sepala', 'largura_petala', 'especie']]
```

	largura_sepala	largura_petala	especie
0	3.5	0.2	Iris-setosa
1	3.0	0.2	Iris-setosa
2	3.2	0.2	Iris-setosa
3	3.1	0.2	Iris-setosa
4	3.6	0.2	Iris-setosa
...	...	...	...
145	3.0	2.3	Iris-virginica
146	2.5	1.9	Iris-virginica
147	3.0	2.0	Iris-virginica
148	3.4	2.3	Iris-virginica
149	3.0	1.8	Iris-virginica

150 rows × 3 columns

Figura 8.35: Saída.

```
# selecionando um subconjunto de linhas e colunas
iris.loc[7:11, ['largura_sepala', 'largura_petala', 'especie']]
```

	largura_sepala	largura_petala	especie
7	3.4	0.2	Iris-setosa
8	2.9	0.2	Iris-setosa
9	3.1	0.1	Iris-setosa
10	3.7	0.2	Iris-setosa
11	3.4	0.2	Iris-setosa

Figura 8.36: Saída.

```
# selecionando linhas segundo um critério
iris[iris['largura_sepala'] > 3.5]
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa
21	5.1	3.7	1.5	0.4	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
44	5.1	3.8	1.9	0.4	Iris-setosa
46	5.1	3.8	1.6	0.2	Iris-setosa
48	5.3	3.7	1.5	0.2	Iris-setosa
109	7.2	3.6	6.1	2.5	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
131	7.9	3.8	6.4	2.0	Iris-virginica

Figura 8.37: Saída.

```
# exibindo a média de cada atributo, agrupado por espécie
iris.groupby('especie').mean()
```

especie	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala
Iris-setosa	5.006	3.418	1.464	0.244
Iris-versicolor	5.936	2.770	4.260	1.326
Iris-virginica	6.588	2.974	5.552	2.026

Figura 8.38: Saída.

Trabalhando com Matplotlib

Agora que já exploramos um pouco os nossos dados com o auxílio da Estatística Descritiva e da biblioteca Pandas, vamos examinar o *dataset Iris* graficamente com a biblioteca Matplotlib. O primeiro tipo de gráfico que criaremos será o gráfico conhecido como *scatter plot*, no qual são marcados os pontos de cruzamento entre duas variáveis. Para isso, será necessário importar o módulo *pyplot* do Matplotlib.

```
# importação do pyplot
import matplotlib.pyplot as plt

# plotando o gráfico de comprimento x largura da sépala
plt.scatter(iris['comprimento_sepala'], iris['largura_sepala'])

# incluindo título do gráfico e rótulos dos eixos
plt.title('Iris: comprimento x largura da sépala')
plt.xlabel('comprimento_sepala')
plt.ylabel('largura_sepala');
```

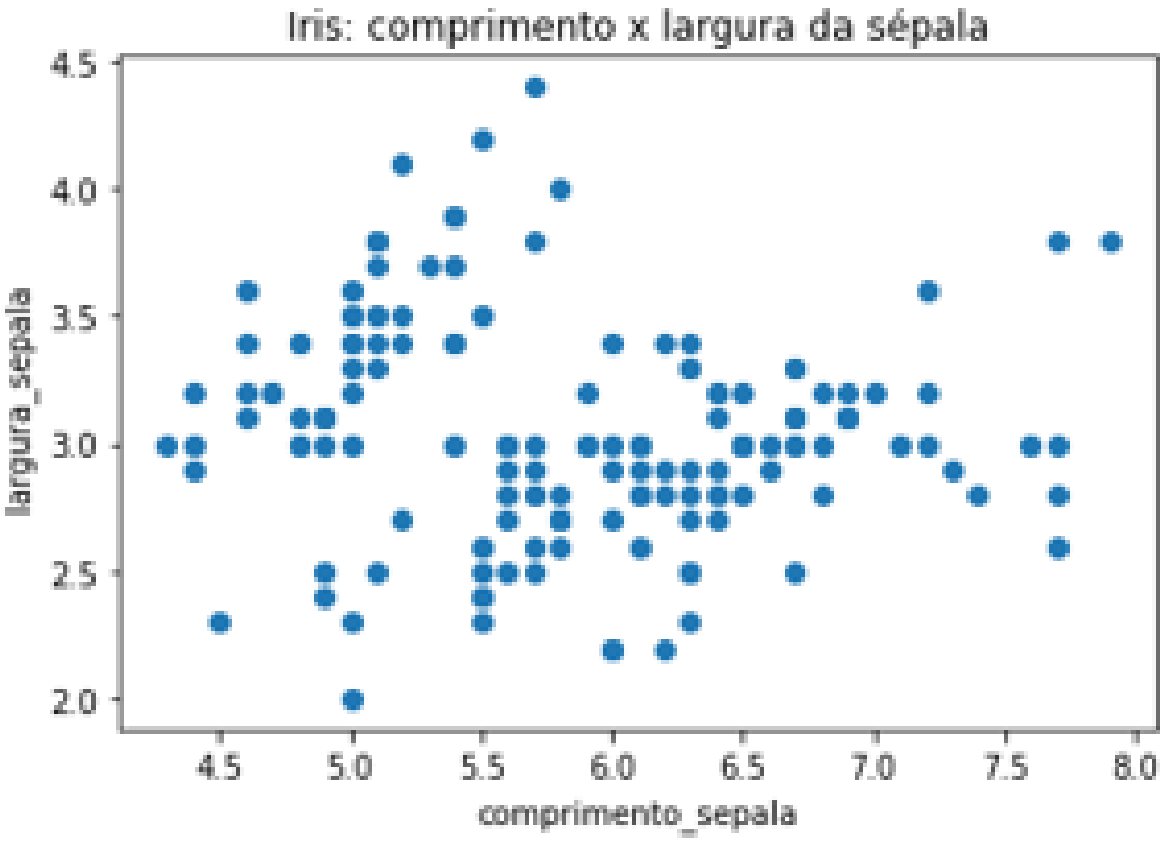


Figura 8.39: Saída.

Se quisermos, também podemos plotar o mesmo gráfico, mas colori-lo de acordo com as espécies das flores (*setosa*, *versicolor* ou *virginica*). Para isso, faremos:

```
# criando o dicionário de cores
cores = {'Iris-setosa':'r', 'Iris-versicolor':'g', 'Iris-virginica':'b'}

# plotando cada um dos pontos de dados
for i in range(len(iris['comprimento_sepala'])):
    plt.scatter(iris['comprimento_sepala'][i], iris['largura_sepala'][i],
color=cores[iris['especie'][i]])

# incluindo título do gráfico e rótulos dos eixos
plt.title('Iris: comprimento x largura da sépala separado por classe')
plt.xlabel('comprimento_sepala')
plt.ylabel('largura_sepala');
```

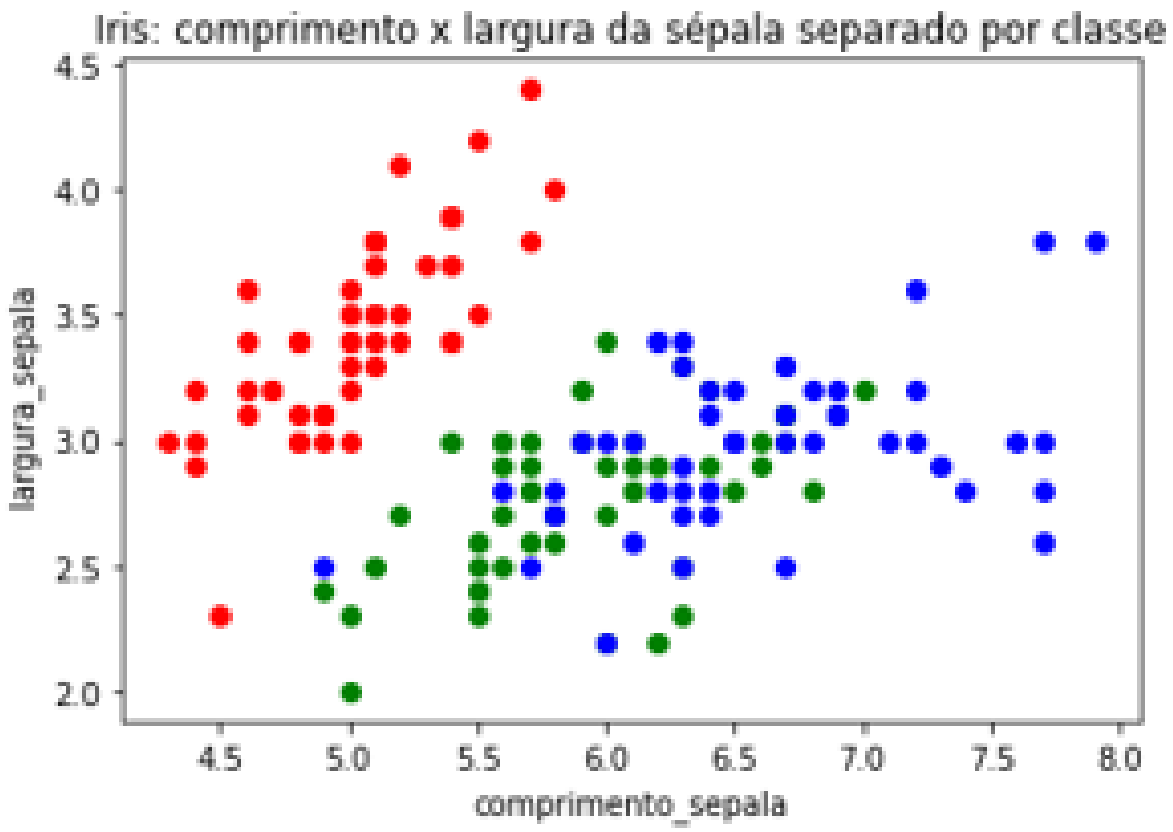


Figura 8.40: Saída.

Também podemos separar as classes usando diferentes marcadores em vez de usar cores (o que é útil em relatórios impressos em preto e branco). Para tal, basta fazer:

```
# criando o dicionário de marcadores
marcadores = {'Iris-setosa':'.', 'Iris-versicolor':'+', 'Iris-
virginica':'x'}

# plotando cada um dos pontos de dados
for i in range(len(iris['comprimento_sepala'])):
   plt.scatter(iris['comprimento_sepala'][i], iris['largura_sepala'][i],
color='blue', marker=marcadores[iris['especie'][i]])

# incluindo título do gráfico e rótulos dos eixos
plt.title('Iris: comprimento x largura da sépala separado por classe')
plt.xlabel('comprimento_sepala')
plt.ylabel('largura_sepala');
```

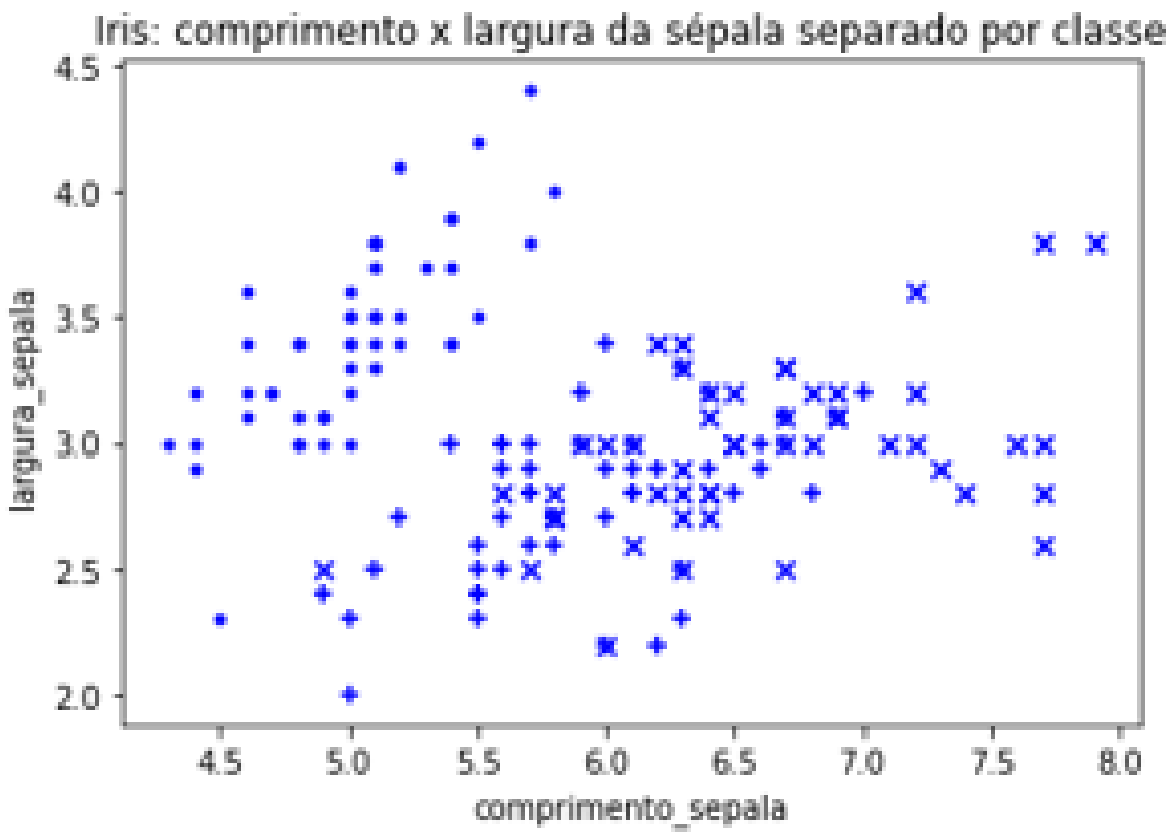


Figura 8.41: Saída.

Vamos agora trabalhar com um tipo diferente de gráfico: o gráfico de linhas (*line plot*). Esse tipo de gráfico é utilizado geralmente quando estamos interessados em expressar variáveis que representam passagem de tempo, ou seja, mostrar a evolução histórica, o que não é o caso deste *dataset*. Mas imagine que o *dataset Iris* esteja representado na ordem em que as amostras foram coletadas ao longo do tempo e queremos visualizar a variação dos atributos dos exemplos coletados ao longo do tempo. Neste caso, faríamos:

```
# Gráfico de Linhas

# selecionando apenas os atributos para serem plotados em um novo
dataframe, sem a classe
atributos = iris.columns.drop(['especie'])
```

```
# criando o eixo x - vai de 0 até a quantidade de linhas do dataset iris -
de 0 a 149
x_data = range(0, iris.shape[0])

# plotando cada atributo
for atributo in atributos: # esse for percorre a lista:
['comprimento_sepala', 'largura_sepala', 'comprimento_petala',
'largura_petala']
    plt.plot(x_data, iris[atributo], label=atributo)

# incluindo título do gráfico e legenda
plt.title('Variação dos atributos do dataset Iris')
plt.legend();
```

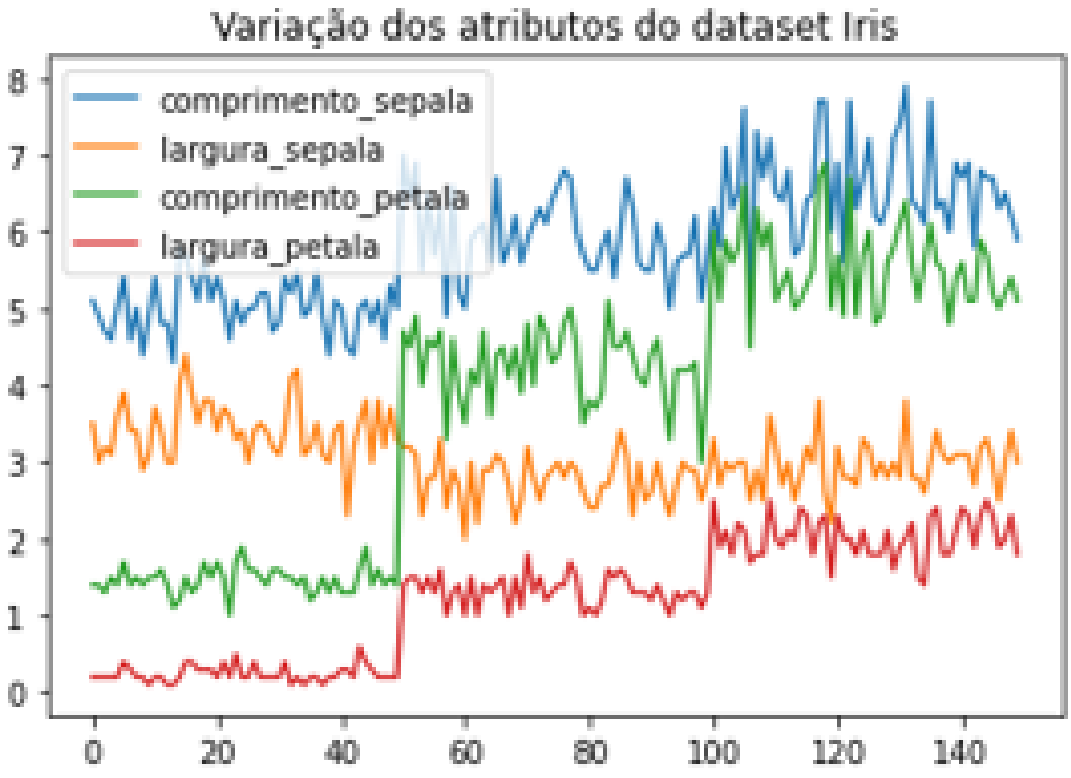


Figura 8.42: Saída.

O histograma exibe a distribuição de frequências de uma determinada variável. É possível informarmos como parâmetro (entre muitas outras possibilidades) o número de *bins* (colunas), que

queremos exibir. O exemplo de código a seguir cria um histograma para o atributo comprimento da sépala com 7 *bins*:

```
# Histograma

# plotando o histograma
plt.hist(iris['comprimento_sepala'], bins=7, edgecolor='black',
color='yellow')

# incluindo título do gráfico e legenda
plt.title('Distribuição de comprimento da sépala')
plt.ylabel('Frequência')
plt.xlabel('comprimento_sepala');
```

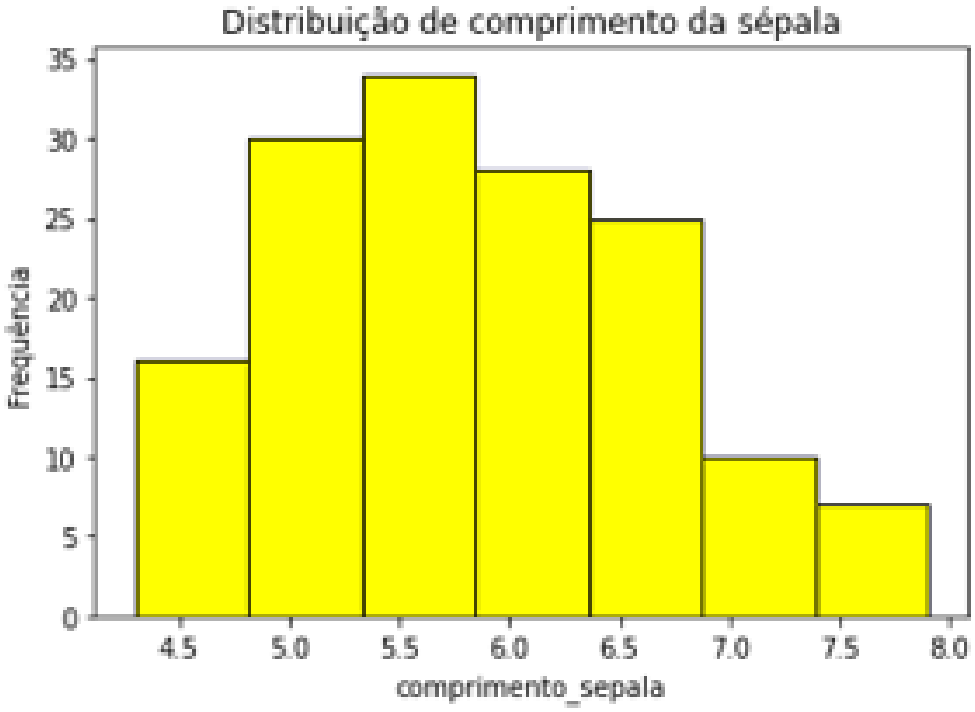


Figura 8.43: Saída.

Por sua vez, o gráfico de barras (*bar plot*) apresenta retângulos (barras), com uma das dimensões proporcional à quantidade a ser representada e a outra arbitrária, mas igual para todas as barras. Os exemplos a seguir ilustram gráficos de barras (o primeiro, com barras verticais, também chamado de gráfico de colunas; e o

segundo, com barras horizontais) para um *dataset* simples, que representa a profissão dos respondentes de uma pesquisa fictícia.

```
# Atribuindo os valores de x e y
profissoes = ['Engenheiro', 'Médico', 'Professor', 'Vendedor',
'Administrador', 'Outros']
quantidade = [15, 30, 25, 28, 33, 39]

# Gráfico de barras

# criando o gráfico de colunas
plt.bar(profissoes, quantidade)

# incluindo título do gráfico e legenda
plt.title("Profissão dos respondentes")
plt.xlabel("Profissão")
plt.ylabel("Número de respondentes");
```

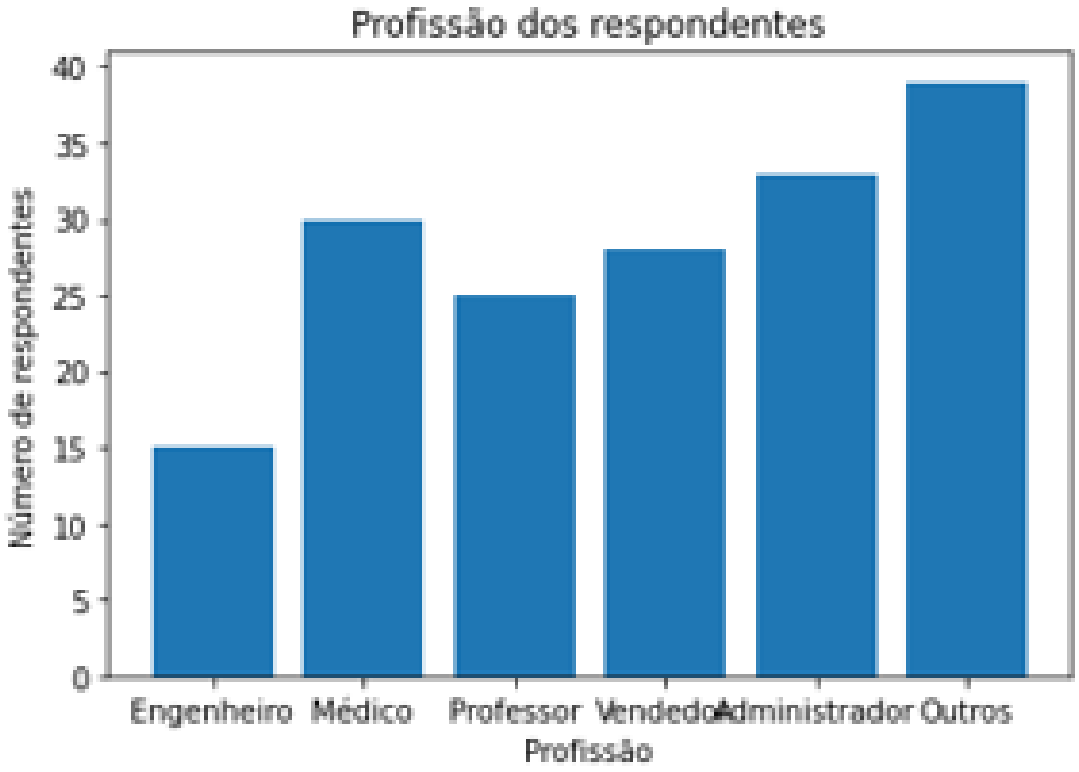


Figura 8.44: Saída.

```
# Gráfico de barras horizontal

# criando o gráfico de barras
plt.barh(profissoes, quantidade)

# incluindo título do gráfico e legenda
plt.title("Profissão dos respondentes")
plt.xlabel("Número de respondentes")
plt.ylabel("Profissão");
```

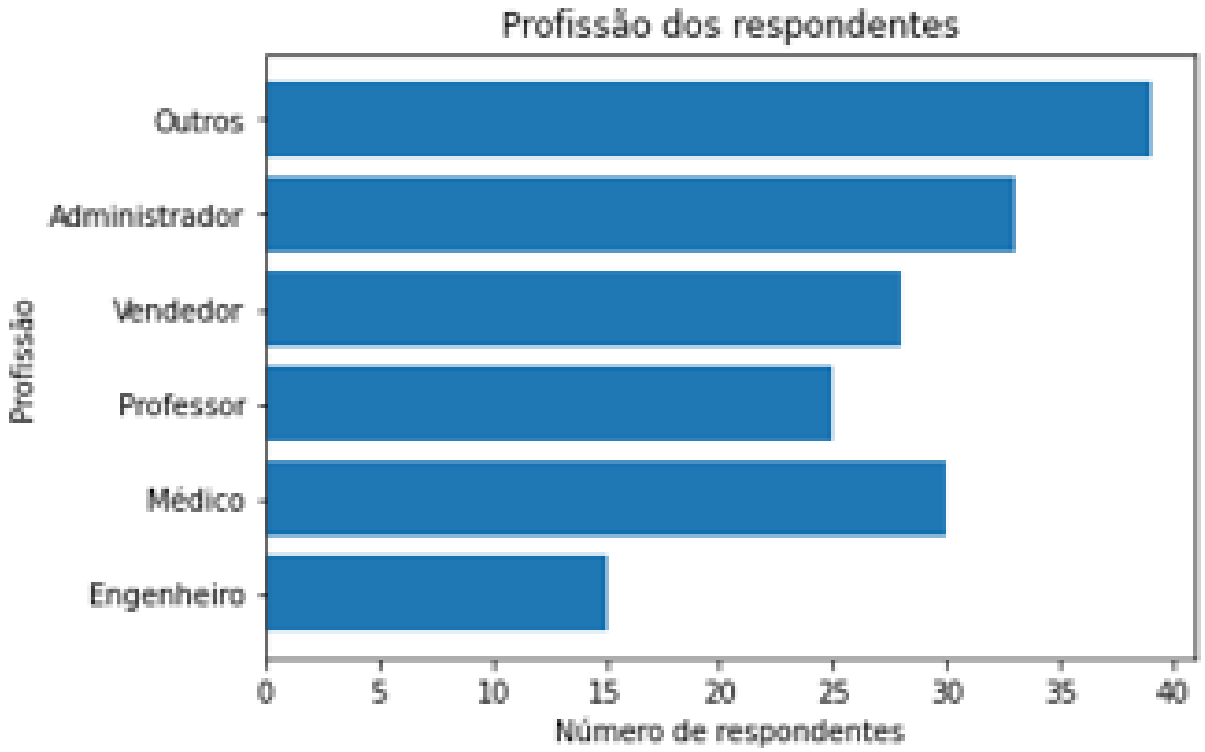


Figura 8.45: Saída.

Outra visualização interessante para quantidades de uma variável qualitativa é o gráfico de pizza (*pie plot*). Vejamos este gráfico com os mesmos dados do exemplo anterior:

```
# criando o gráfico de pizza
plt.pie(quantidade, labels = profissoes);
```

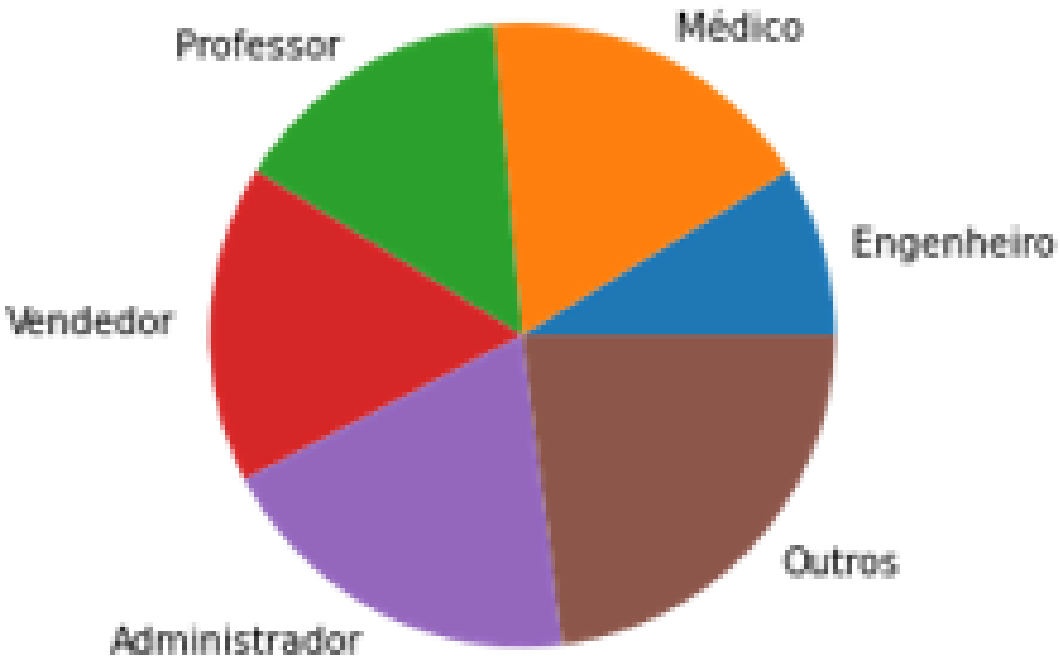


Figura 8.46: Saída.

Para finalizar os exemplos da biblioteca Matplotlib, vamos plotar, em uma única figura, 4 histogramas do *dataset Iris*, um para cada atributo:

```
# configurando um grupo de subplots em um grid de 1 linha e 2 colunas
# e também o tamanho da figura
fig, ax = plt.subplots(2,2, figsize = (8, 8))

# subplot1: histograma do comprimento_sepala
ax[0,0].hist(iris['comprimento_sepala'], bins=7, edgecolor='black')
ax[0,0].set_title("comprimento_sepala")

# subplot2: histograma da largura_sepala
ax[0,1].hist(iris['largura_sepala'], bins=7, edgecolor='black')
ax[0,1].set_title("largura_sepala")

# subplot3: histograma do comprimento_petala
ax[1,0].hist(iris['comprimento_petala'], bins=7, edgecolor='black')
ax[1,0].set_title("comprimento_petala")

# subplot4: histograma da largura_petala
```

```
ax[1,1].hist(iris['largura_petala'], bins=7, edgecolor='black');
ax[1,1].set_title("largura_petala");
```

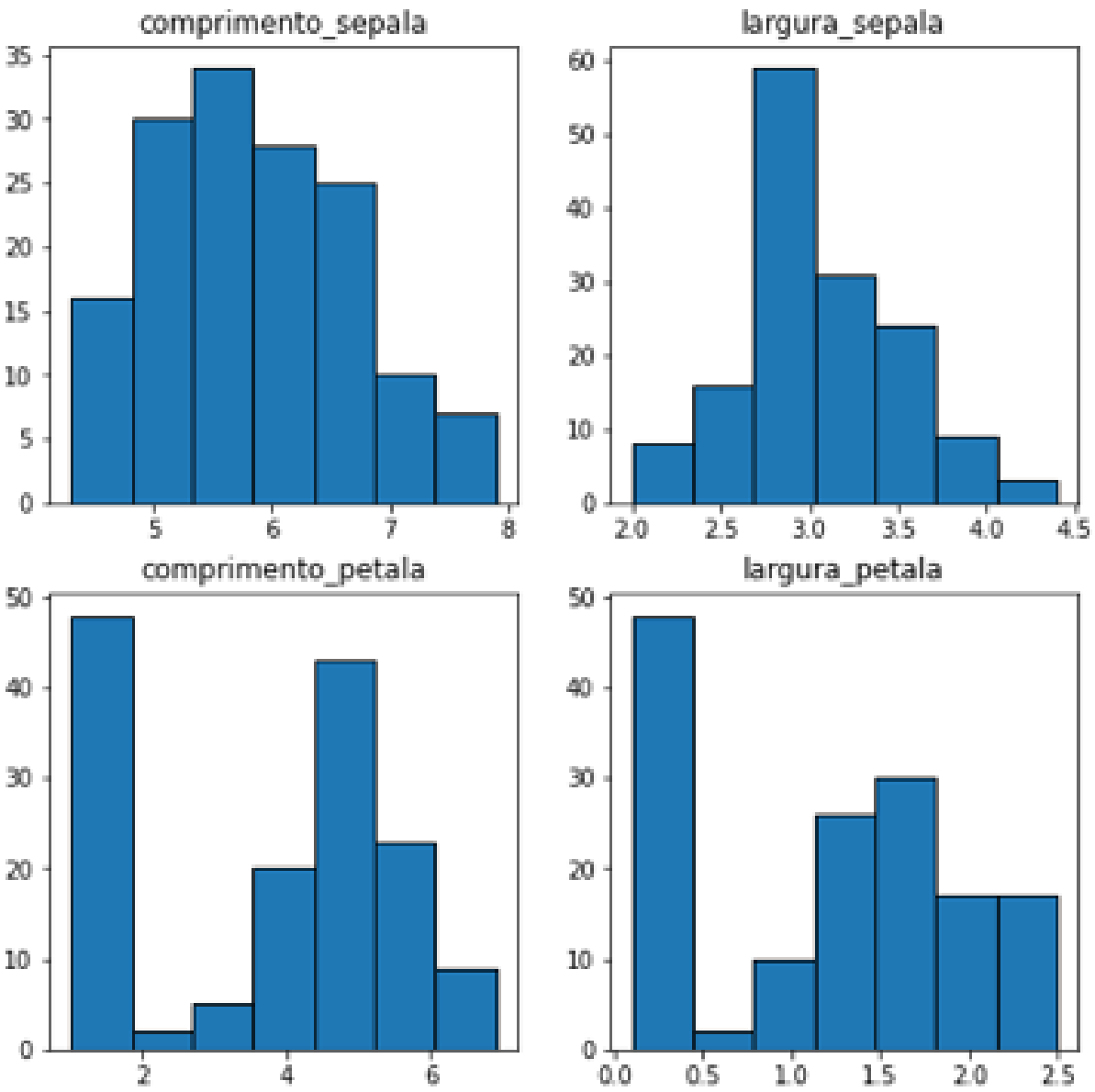


Figura 8.47: Saída.

Poderíamos fazer o mesmo do exemplo anterior, substituindo os histogramas por *boxplots*:

```
# configurando um grupo de subplots em um grid de 1 linha e 2 colunas
# e também o tamanho da figura
fig, ax = plt.subplots(2,2, figsize = (8, 8))
```

```
# subplot1: boxplot do comprimento_sepala
ax[0,0].boxplot(iris['comprimento_sepala'])
ax[0,0].set_title("comprimento_sepala")

# subplot2: boxplot da largura_sepala
ax[0,1].boxplot(iris['largura_sepala'])
ax[0,1].set_title("largura_sepala")

# subplot3: boxplot do comprimento_petala
ax[1,0].boxplot(iris['comprimento_petala'])
ax[1,0].set_title("comprimento_petala")

# subplot4: boxplot da largura_petala
ax[1,1].boxplot(iris['largura_petala']);
ax[1,1].set_title("largura_petala");
```

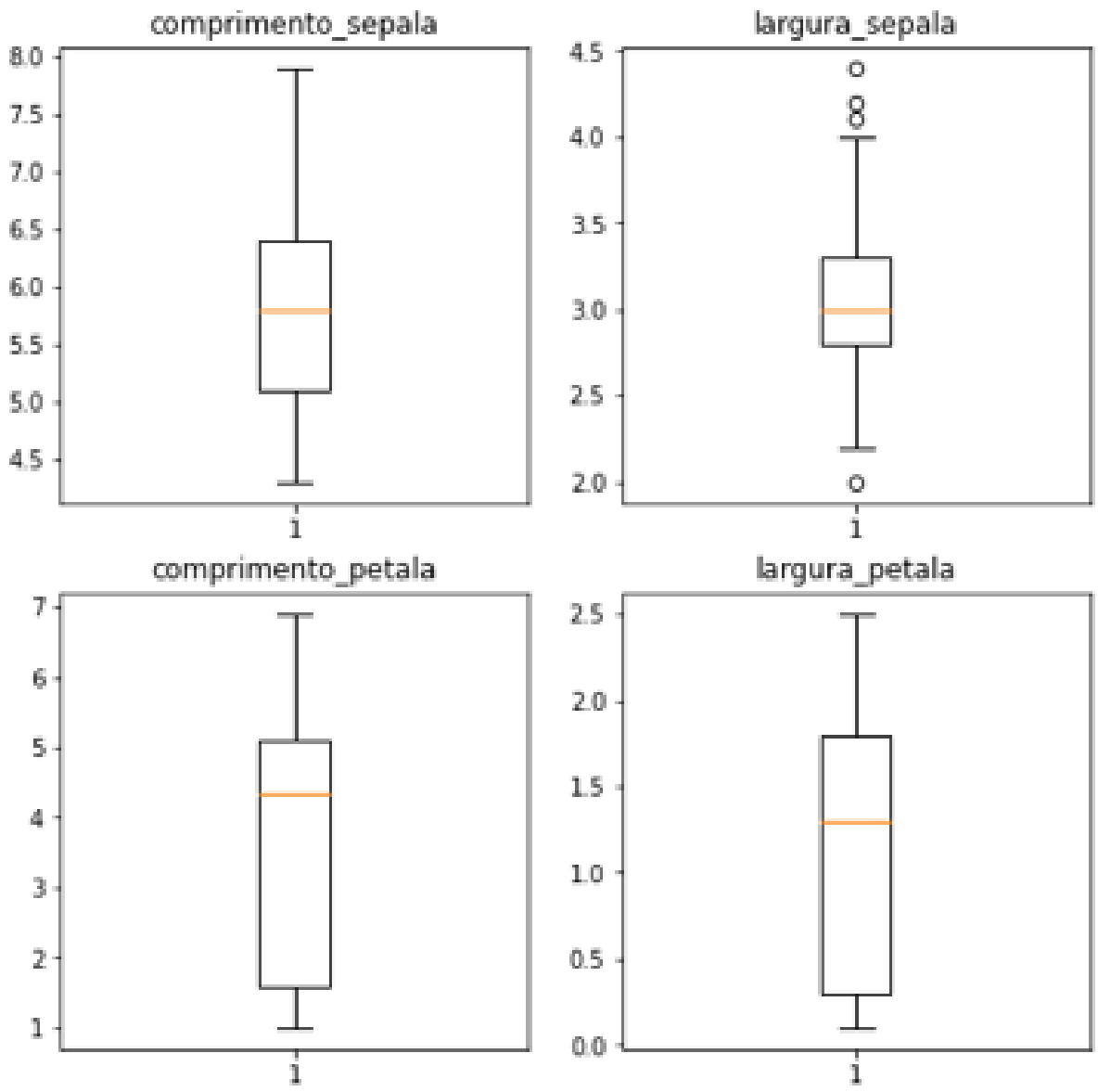


Figura 8.48: Saída.

É importante ressaltar que a biblioteca Matplotlib é muito extensa e seus gráficos são altamente configuráveis. Aqui procuramos trazer apenas exemplos simples para facilitar o entendimento dos códigos. Entretanto, praticamente tudo pode ser customizado, desde configurações mais simples, como cores, tamanhos ou exibição de *labels*, e até outras personalizações mais avançadas. Recomendamos que você procure essas opções na documentação oficial de acordo com a sua necessidade de uso.

Trabalhando com Seaborn

Vamos agora explorar outra biblioteca para gráficos em Python: o pacote Seaborn. Todos os exemplos a seguir também utilizarão o *dataset Iris*, que já carregamos previamente. Você notará que os três primeiros exemplos de gráficos com Seaborn produzem o mesmo resultado de alguns gráficos que criamos com Matplotlib. Vamos começar importando a biblioteca:

```
# importação do seaborn
import seaborn as sns
```

Agora vamos plotar um *scatter plot* simples, em seguida, o mesmo, colorido por espécies, e um gráfico de linhas com os atributos do *dataset Iris*. Repare que são necessárias bem menos linhas de código em comparação com o Matplotlib:

```
# scatter plot com Seaborn
sns.scatterplot(x='comprimento_sepala', y='largura_sepala', data=iris);
```

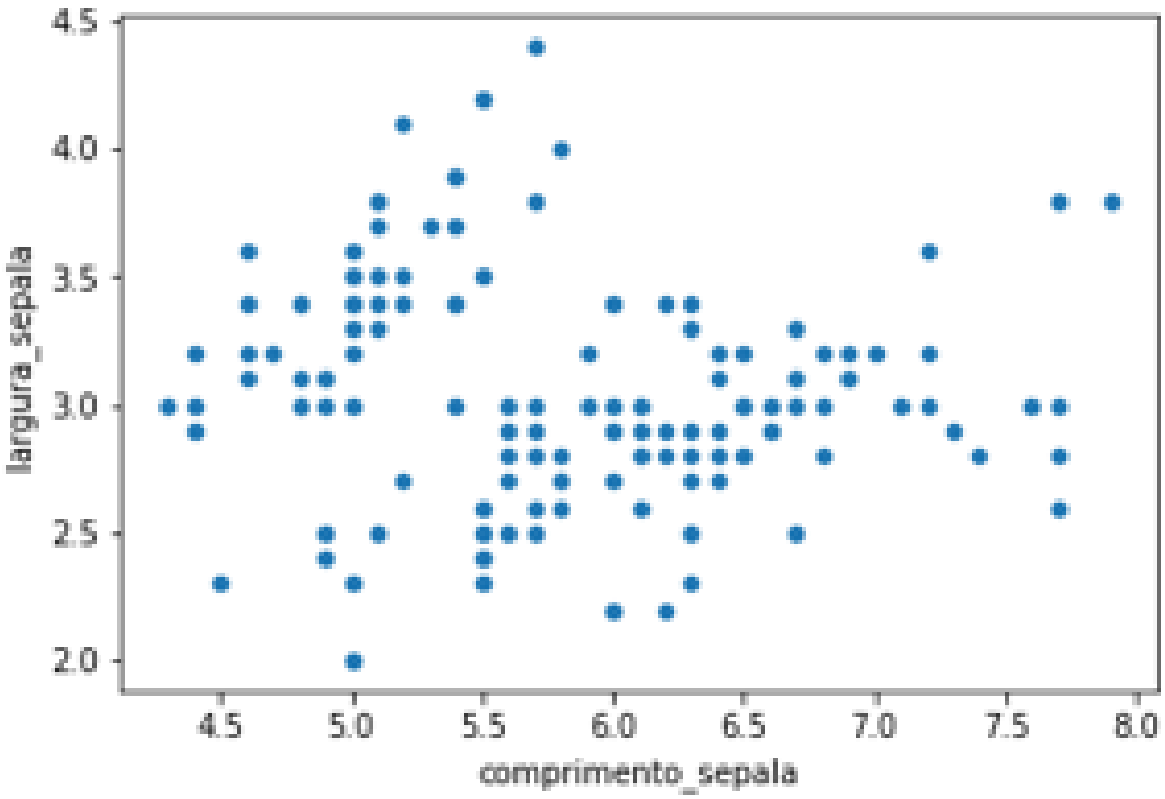


Figura 8.49: Saída.

```
# scatter plot com Seaborn colorido por classes
sns.scatterplot(x='comprimento_sepala', y='largura_sepala', hue='especie',
data=iris);
```

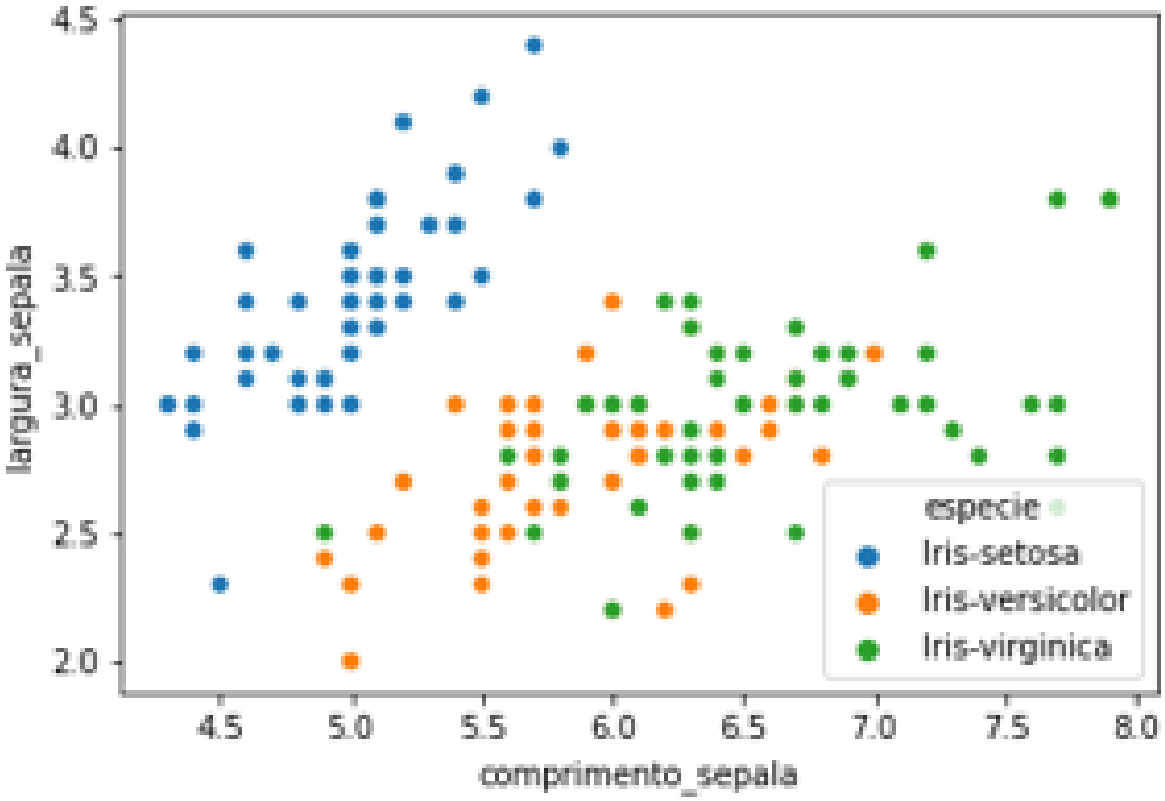


Figura 8.50: Saída.

```
# gráfico de linhas com Seaborn
sns.lineplot(data=iris.drop(['especie'], axis=1));
```

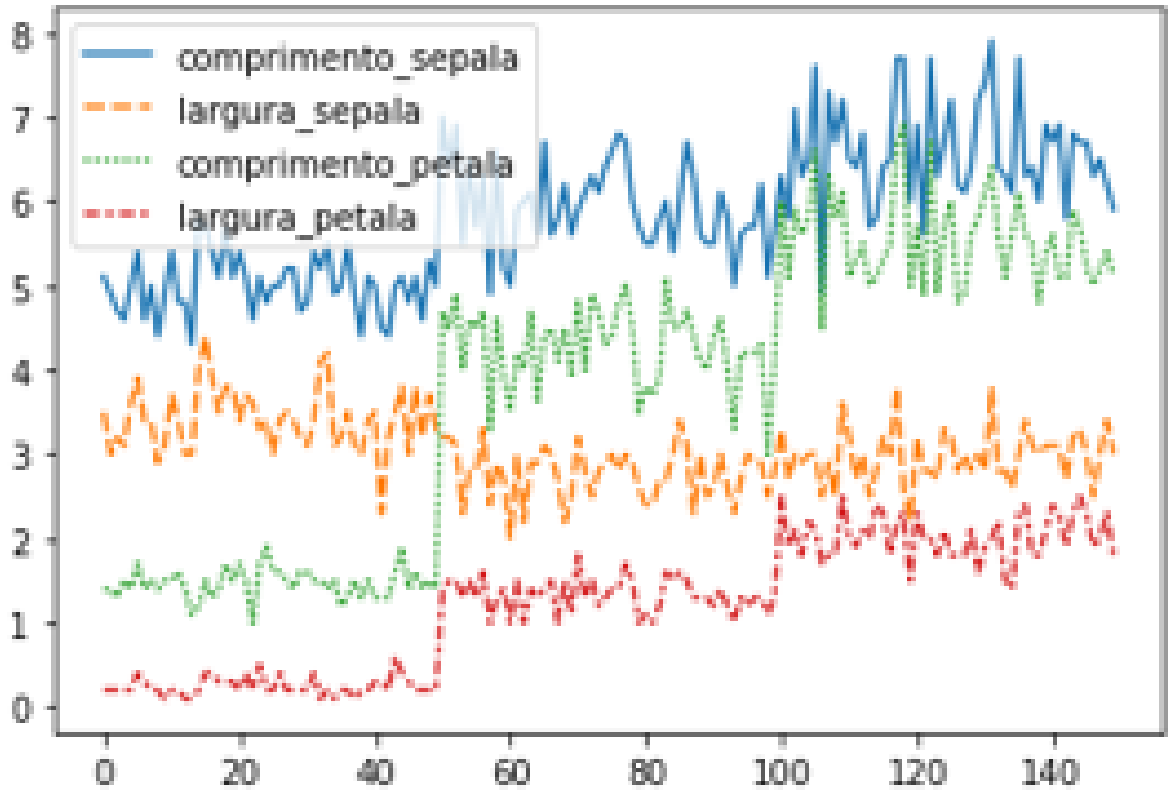


Figura 8.51: Saída.

Vejamos um exemplo do gráfico *boxplot* com a biblioteca Seaborn, exibindo os *boxplots* de todos os atributos em um mesmo gráfico:

```
# boxplot com Seaborn
sns.boxplot(data = iris);
```

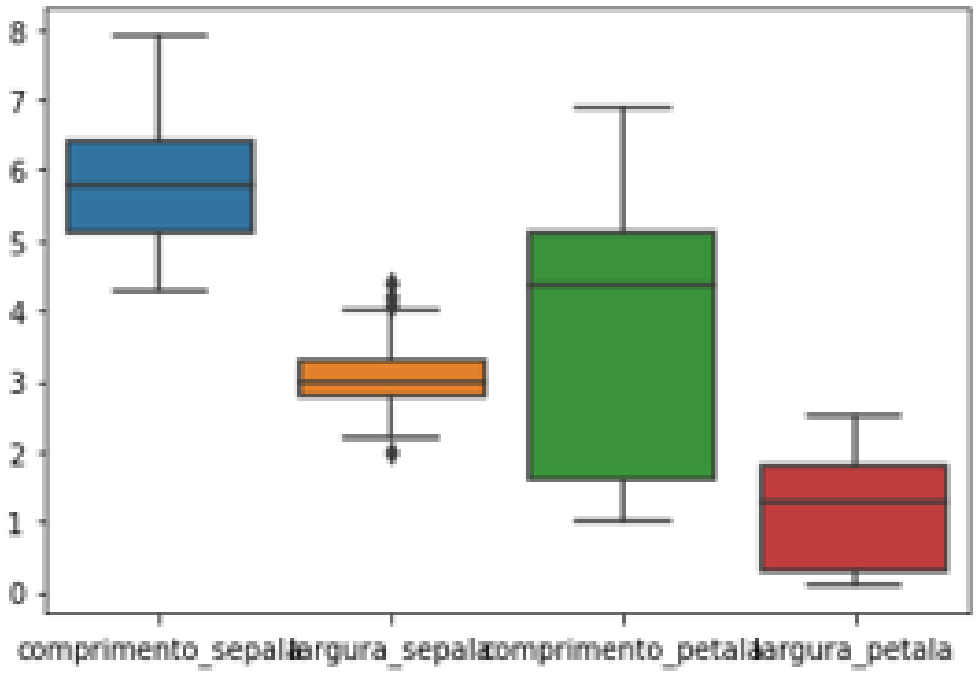


Figura 8.52: Saída.

Agora vamos plotar um histograma do comprimento da sépala:

```
# histograma com seaborn
sns.histplot(iris['comprimento_sepala'], bins=7);
```

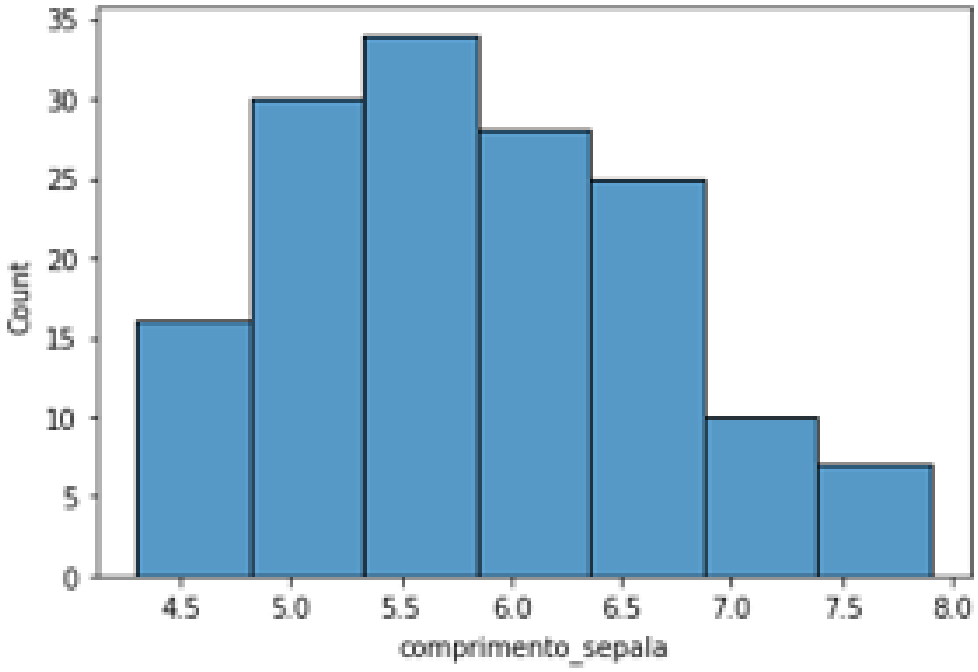


Figura 8.53: Saída.

Podemos também plotar *boxplots* simples facilmente:

```
# boxplot com seaborn
sns.boxplot(x=iris['comprimento_sepala']);
```

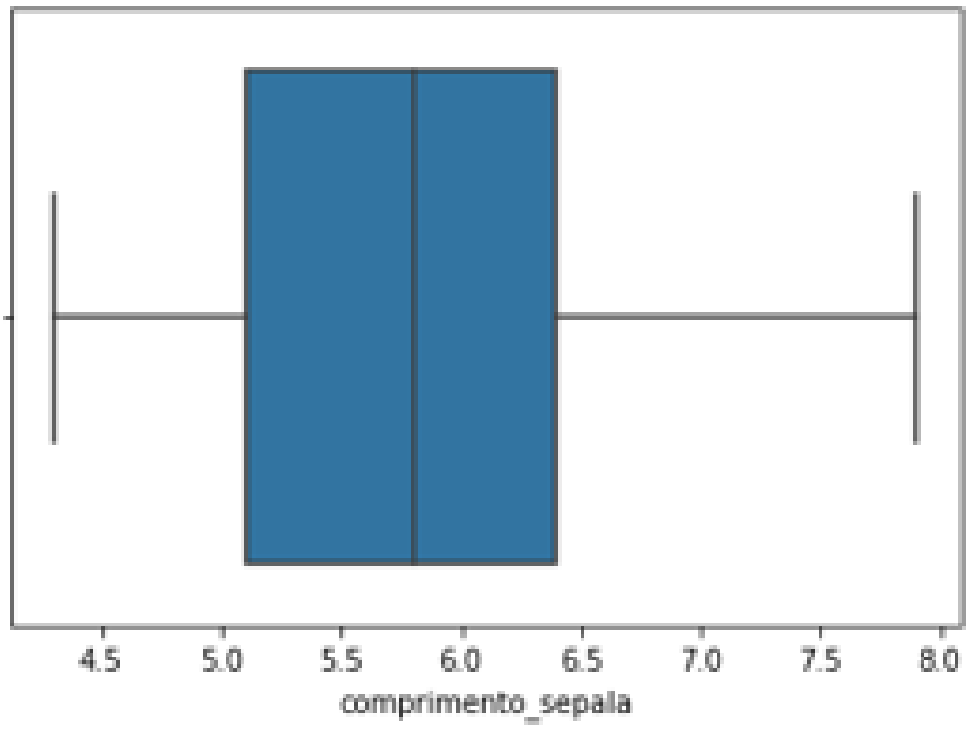


Figura 8.54: Saída.

Um gráfico muito útil para a análise de dados é o **gráfico de correlação**, que exibe graficamente as correlações entre os atributos numéricos de um *dataset*. Com Seaborn, basta uma linha de código para plotá-la. Para fazer algo similar com Matplotlib, são necessárias diversas linhas de código.

```
# gráfico de correlação com Seaborn
sns.heatmap(iris.corr(), annot=True, cmap='RdBu', vmin=-1, vmax=1);
```

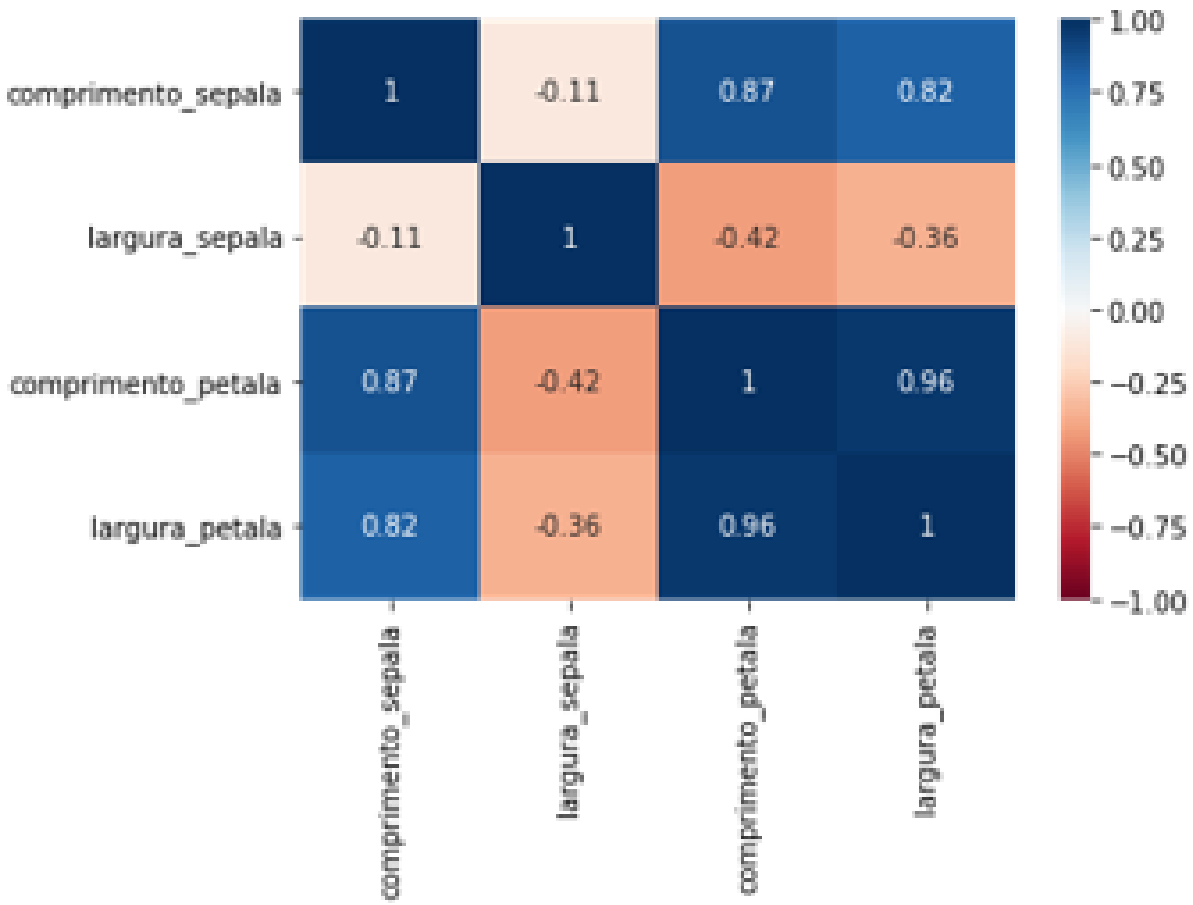


Figura 8.55: Saída.

Para finalizar, vamos plotar um gráfico composto muito interessante que apresenta, em uma só figura, *scatter plots* de cada par de atributos numéricos do *dataset* e o histograma do atributo na diagonal principal. Se configurarmos o parâmetro *hue*, em vez do histograma, o gráfico exibirá na diagonal principal a distribuição de cada uma das espécies pelo atributo, usando *density plots*. Os dois exemplos são exibidos a seguir. Repare que basta uma única linha de código para que este gráfico tão informativo seja exibido.

```
# pair plot 1 com Seaborn
sns.pairplot(iris);
```

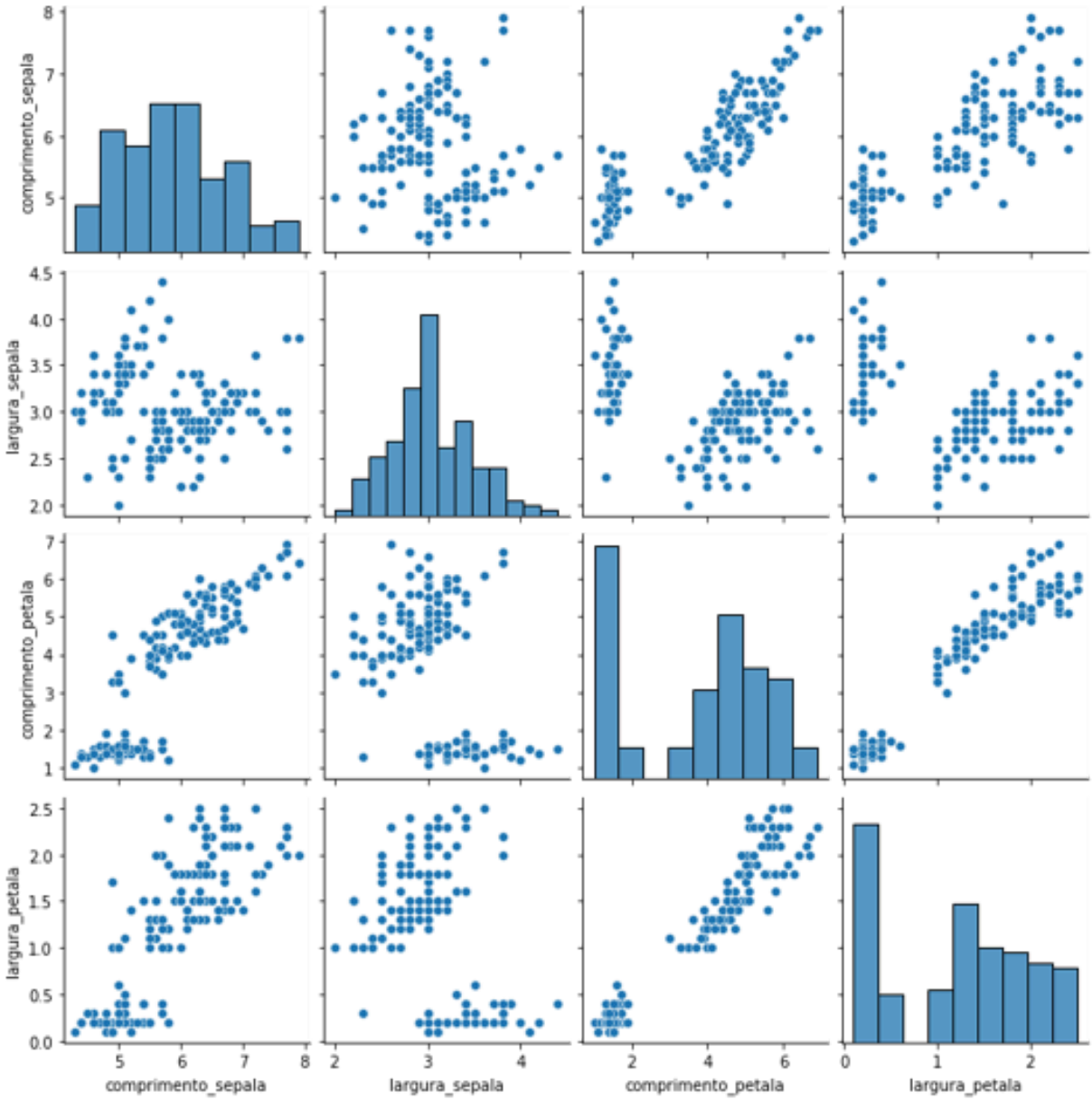


Figura 8.56: Saída.

```
# pair plot 2 com Seaborn
sns.pairplot(iris, hue = "especie");
```

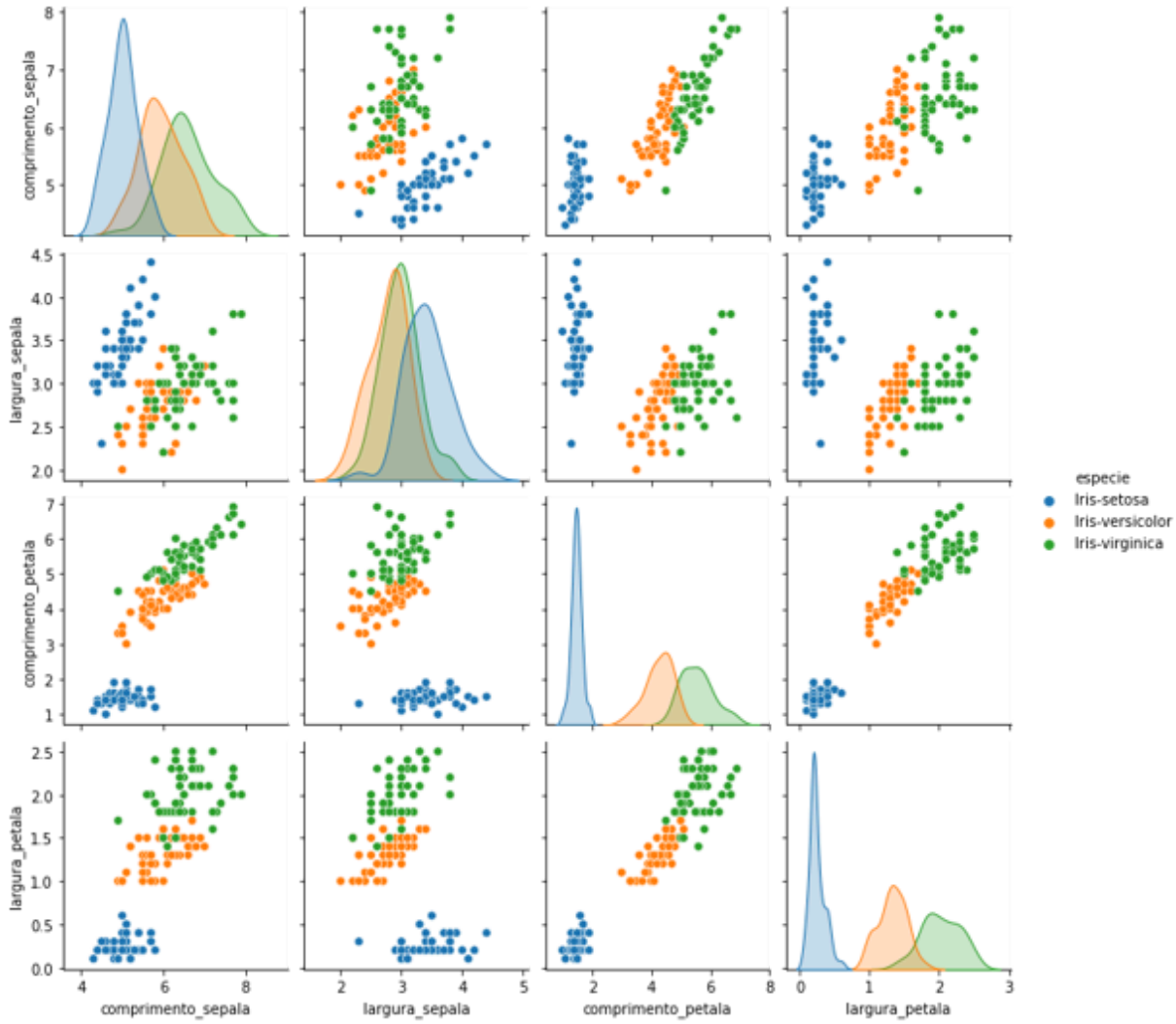


Figura 8.57: Saída.

Assim como a Matplotlib, a biblioteca Seaborn é muito extensa e seus gráficos podem ser personalizados de acordo com sua necessidade. Recomendamos que você explore mais exemplos na documentação oficial de acordo com a sua necessidade de uso.

Capítulo 9
Pré-processamento de dados

No capítulo anterior, estudamos como fazer uma análise exploratória de dados e aprendemos a exibir nossos dados em forma gráfica. Após coletar e analisar os dados na etapa anterior, é necessário limpar e transformar adequadamente os seus dados, a fim de obter na próxima etapa os melhores resultados possíveis nos algoritmos de *Machine Learning*, ou simplesmente apresentar dados mais confiáveis para os clientes em soluções de *Business Intelligence*. Neste capítulo, vamos focar na etapa de **pré-processamento de dados**, como ilustra a figura a seguir:

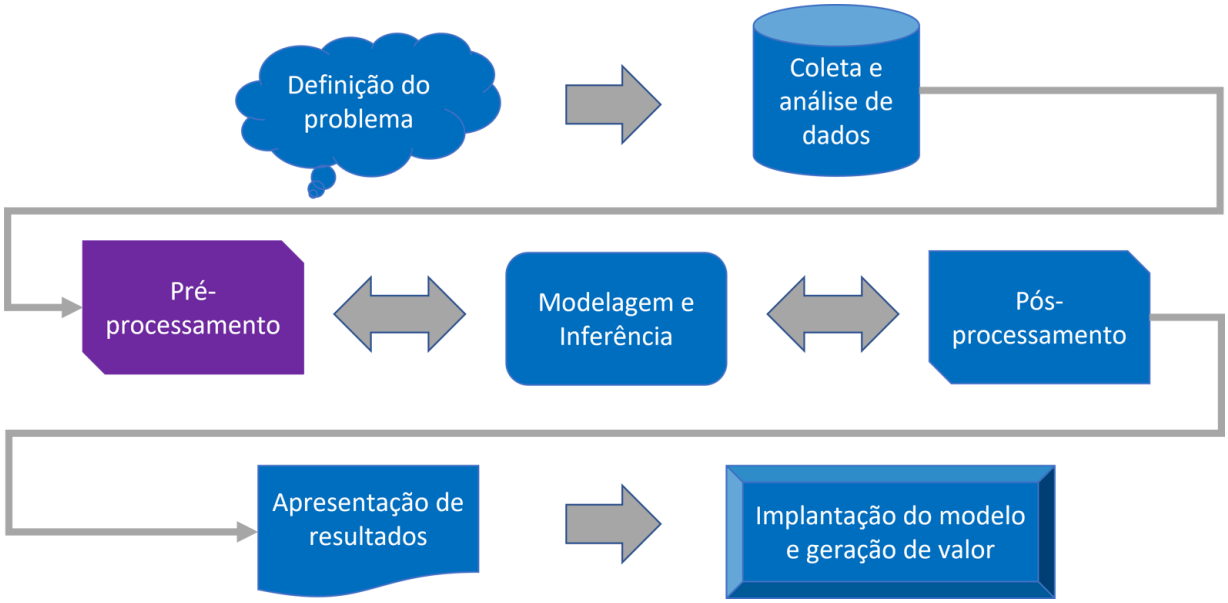


Figura 9.1: Esquema básico de um projeto de Ciência de Dados – Etapa de pré-processamento de dados.

Tendo examinado os seus dados com cuidado, você pode identificar alguns problemas que devem ser tratados nesta etapa. Para isso, é importante verificar questões como:

- **Unidades de medida**: observar se as variáveis estão sendo medidas na mesma unidade (idade medida em meses e em

anos, renda em dólares e em reais, etc.) para fazer as devidas conversões, se necessário.

- **Valores faltantes (*missings*)**: verificar se um atributo tem muitos registros faltantes. Neste caso, ele não deve ser utilizado como entrada de um modelo sem um tratamento apropriado, por exemplo, preenchendo os *missings* com valores adequados.
- **Valores inconsistentes**: entender se os valores inconsistentes são valores inválidos ou *outliers*. Temos valores inválidos quando temos um valor negativo em um campo que só pode ser positivo, ou um texto quando se espera um número. Já *outliers* são valores fora do intervalo esperado.
- **Intervalo dos valores**: prestar atenção no intervalo dos valores das variáveis. Atributos como salário podem conter valores de 0 a mais de meio milhão de reais, por exemplo. Esse grande intervalo pode ser um problema para alguns modelos, e pode ser necessário transformar essa coluna usando uma transformação logarítmica para reduzir o seu intervalo de valores. Já variáveis com todos os valores em intervalos pequenos (exemplo: idades entre 50 e 55) provavelmente não serão uma informação útil para os modelos. Vale lembrar que o intervalo de valores adequado varia de acordo com o domínio da aplicação.

Para ilustrar a importância da etapa de pré-processamento de dados, vamos examinar um *dataset* simplificado do Banco de Crédito Alemão, adaptado do *UCI Machine Learning Repository* (https://archive.ics.uci.edu/). A figura a seguir ilustra um trecho desses dados:

	custid	sex	is.employed	income	marital.stat	health.ins
1	2068	F	*NA*	11300	Married	TRUE
2	2073	F	*NA*	0	Married	TRUE
3	2848	M	TRUE	4500	Never Married	FALSE
4	5641	M	TRUE	20000	Never Married	FALSE
5	6369	F	TRUE	12000	Never Married	TRUE
6	8322	F	TRUE	180000	Never Married	TRUE
7	8521	M	TRUE	120000	Never Married	TRUE

	housing.type	recent.move	num.vehicles	age	state.of.res
1	Homeowner free and clear	FALSE	2	49	Michigan
2	Rented	TRUE	3	40	Florida
3	Rented	TRUE	3	22	Georgia
4	Occupied with no rent	FALSE	0	22	New Mexico
5	Rented	TRUE	1	31	Florida
6	Homeowner with mortgage/loan	FALSE	1	40	New York
7	Homeowner free and clear	TRUE	1	39	Idaho

Figura 9.2: Trecho do dataset do Banco de Crédito Alemão.

Vamos examinar os dados sumarizados para verificar a necessidade de operações de pré-processamento. Observando a figura a seguir, nota-se que 1/3 dos dados da coluna *is.employed* (que mapeia se a pessoa está ou não empregada) está categorizado como *NA* (*not available*), ou seja, com valores faltantes. São possíveis opções de tratamento para essa coluna: realizar a completa exclusão do *dataset*; ou substituir os valores *FALSE*, *TRUE* e *NA* respectivamente por *NÃO*, *SIM* e *DESCONHECIDO*.

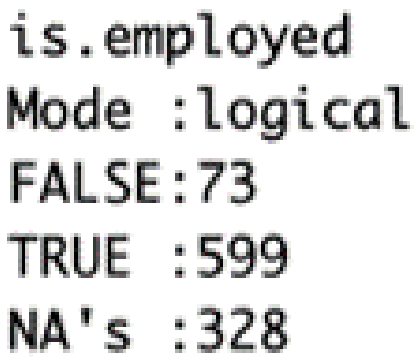


Figura 9.3: Análise do dataset do Banco de Crédito Alemão.

Agora examinemos a coluna *income* (renda). Notamos que há valores de renda negativos (o mínimo é -8700), que são potencialmente inválidos. Terá ocorrido um erro na entrada desses dados, ou a renda negativa significa que o cliente tem débitos, tais como empréstimos ou financiamentos? São possíveis opções para o tratamento dessa coluna: simplesmente excluir do *dataset* as linhas com *income* negativo; ou substituir esses valores por zero.

Figura 9.4: Análise do dataset do Banco de Crédito Alemão.

Outro potencial problema da coluna *income* é que seu intervalo de valores (*range*) é grande demais. Isso pode afetar negativamente alguns modelos de *Machine Learning*, então pode ser uma boa ideia

aplicar uma transformação logarítmica nesses dados para reduzir o *range* de valores.

Analisando as colunas *housing.type*, *recent.move* e *num.vehicles*, podemos notar 56 linhas com valores faltantes (*missings*), representados pelo valor *NA*. Como são poucas linhas em relação ao tamanho total do *dataset* (5,6% do total), podemos simplesmente excluí-las, ou então substituir os valores faltantes pela média (ou mediana) dos demais valores da coluna.

```
                              housing.type
Homeowner free and clear    :157
Homeowner with mortgage/loan:412
Occupied with no rent       : 11
Rented                      :364
NA's                        : 56
```

Figura 9.5: Análise do dataset do Banco de Crédito Alemão.

```
recent.move       num.vehicles
Mode :logical   Min.   :0.000
FALSE:820       1st Qu.:1.000
TRUE :124       Median :2.000
NA's :56        Mean   :1.916
                3rd Qu.:2.000
                Max.   :6.000
                NA's   :56
```

Figura 9.6: Análise do dataset do Banco de Crédito Alemão.

Finalmente, ao analisar a coluna *age*, os valores de mínimo (0.0) e máximo (146.7) chamam a atenção. É possível que o banco tenha clientes muito jovens ou muito idosos, mas poderíamos normalizar as idades pela média (se a idade absoluta não for importante para esse problema) ou então converter essa variável numérica em categorias (*bins*).

```
      age
Min.   :  0.0
1st Qu.: 38.0
Median : 50.0
Mean   : 51.7
3rd Qu.: 64.0
Max.   :146.7
```

Figura 9.7: Análise do dataset do Banco de Crédito Alemão.

Em projetos de Ciência de Dados, as operações de pré-processamento são extremamente necessárias para melhorar a qualidade dos dados que trabalhamos, a fim de possibilitar análises corretas e buscar os melhores resultados possíveis nos algoritmos de *Machine Learning* na etapa seguinte. A seguir, veremos com mais detalhes as principais técnicas de pré-processamento de dados.

9.1 Técnicas de pré-processamento de dados

Existem diversas técnicas de pré-processamento que podem ser utilizadas quando estamos preparando um *dataset* para a criação de modelos de *Machine Learning*, ou mesmo análises de *Business Intelligence*. É importante ressaltar que, antes de aplicar as operações de pré-processamento nos dados, recomenda-se fazer

uma cópia dos dados-fontes originais e, após a aplicação das operações de pré-processamento, organizar os dados em um novo único tratado.

Limpeza

De forma resumida, a limpeza consiste na verificação da consistência das informações, correção de possíveis erros de preenchimento ou eliminação de valores desconhecidos, redundantes ou não pertencentes ao domínio.

Quando trabalhamos com dados reais, é muito comum que eles estejam incompletos, ruidosos ou inconsistentes. A limpeza é muito importante, pois, quanto pior a qualidade dos dados, pior será a qualidade dos modelos gerados (*"garbage in, garbage out"*). Vale lembrar que a melhor maneira de evitar a poluição de dados é organizando a entrada dos dados, mas como nem sempre isso é possível, a realização da limpeza dos dados trabalhados ganha mais importância.

Para que a limpeza de dados seja feita de forma adequada, é essencial o envolvimento das pessoas especialistas de negócio. Elas podem, por exemplo, apoiar na limpeza de informações ausentes, ajudando o(a) cientista de dados a realizar exclusão de casos, preenchimento manual, preenchimento com valores globais constantes, preenchimento com medidas estatísticas ou preenchimento através de métodos de *Machine Learning*. Além disso, especialistas podem atuar na limpeza de inconsistências e valores não pertencentes ao domínio, realizando a exclusão de casos ou correção de erros.

A figura a seguir ilustra os principais casos que podem ser tratados com a técnica de limpeza de linhas (instâncias) ou colunas (características).

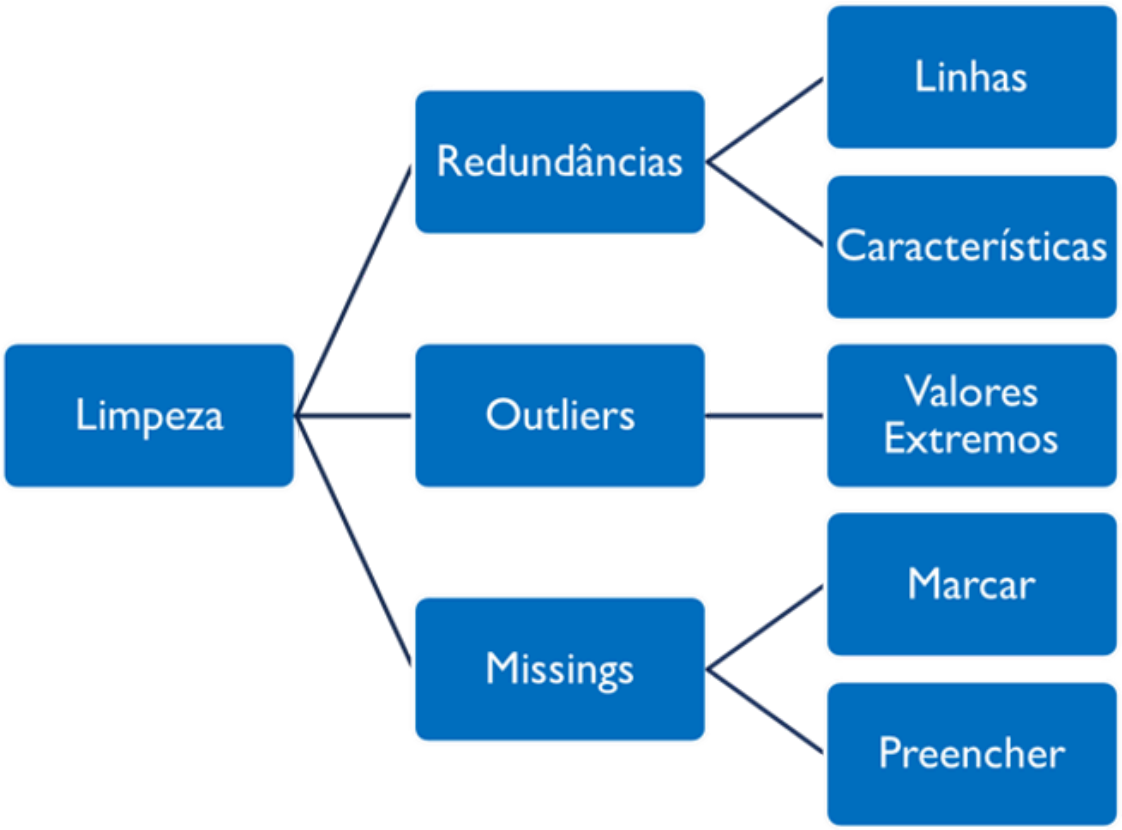


Figura 9.8: Limpeza de dados.

Agregação

A agregação consiste em combinar dois ou mais objetos em um único. Por exemplo, agregar os registros de transações de vendas diárias de produtos em lojas distintas em uma única transação mensal por loja, como ilustra a figura a seguir:

Item	Local	Data	Preço
Relógio	Paris	09/06/2004	100
Pilha	Rio	09/06/2004	3
Vestido	Rio	09/06/2004	50
Sapatos	Paris	09/06/2004	35

Local	Data	Valor
Paris	jun/04	135
Rio	jun/04	53

Figura 9.9: Agregação de dados.

Como vantagens da agregação, podemos citar que um conjunto de dados menor tem menor custo de processamento. Além disso, a agregação pode atuar como uma mudança de granularidade, muito usada em *Business Intelligence* com as operações *OLAP* (*Online Analytical Processing*), permitindo manipular e analisar um grande volume de dados sob múltiplas perspectivas. Como desvantagens, podemos perder detalhes que podem ser interessantes (por exemplo, identificar o dia da semana que tem maiores vendas).

Amostragem

A amostragem consiste em selecionar um subconjunto dos dados a serem analisados ou trabalhados. O princípio-chave da amostragem é a ideia de que usar uma amostra funcionará quase tão bem quanto usar o conjunto inteiro de dados, desde que a amostra seja representativa, ou seja, que tenha aproximadamente a mesma

propriedade de interesse do conjunto original de dados (por exemplo, a média).

São possíveis abordagens a amostragem aleatória simples (que pode ser com/sem substituição) e a amostragem estratificada (que acomoda frequências diferentes). A amostragem aleatória simples e a estratificada são ilustradas pelas figuras a seguir:

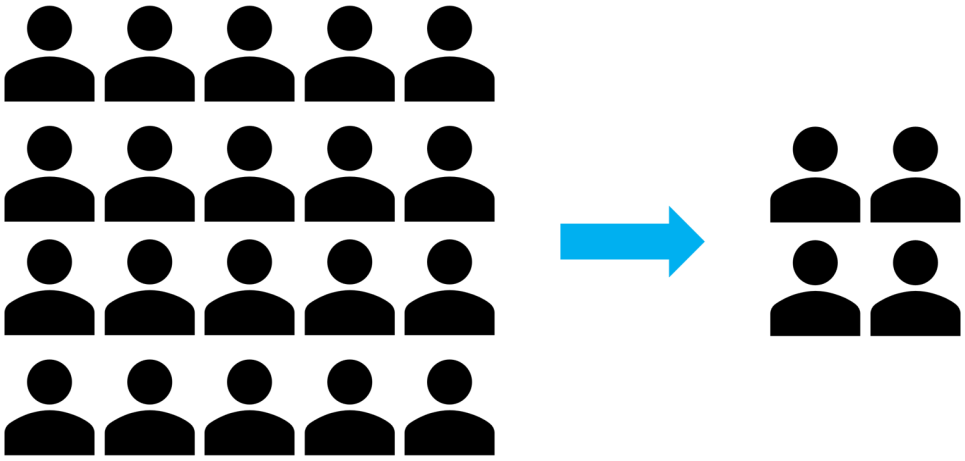


Figura 9.10: Amostragem aleatória simples.

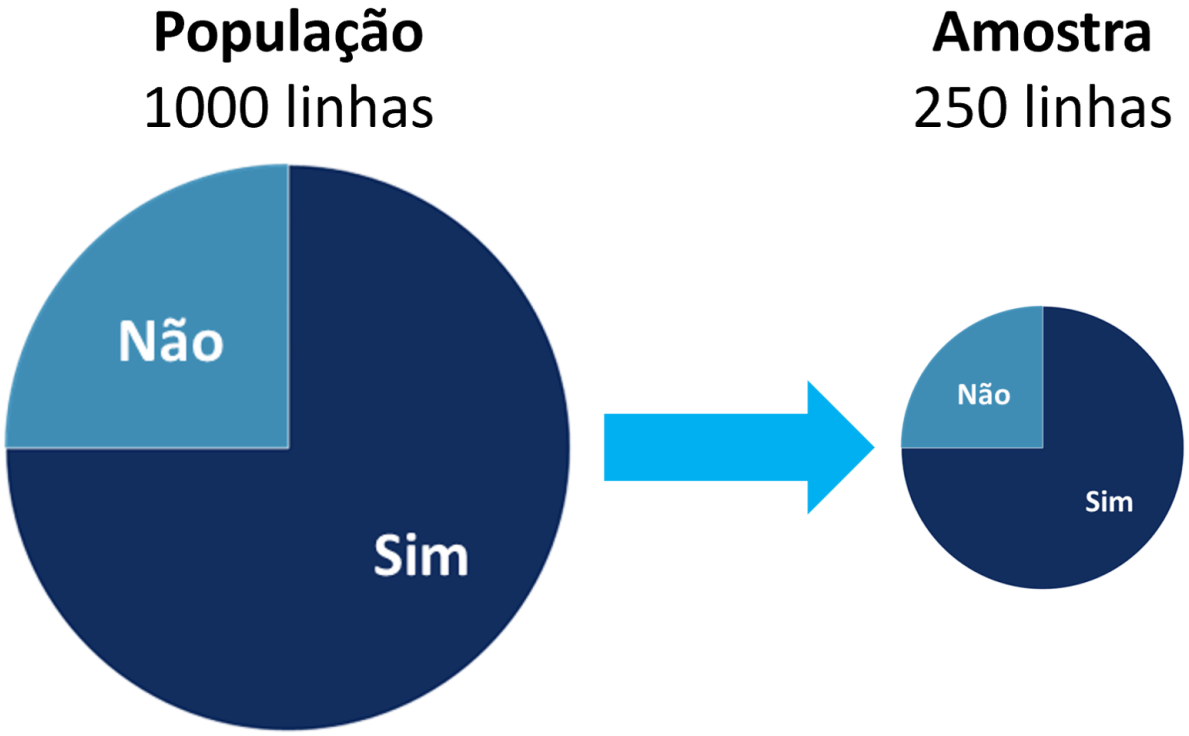


Figura 9.11: Amostragem estratificada.

Em *Machine Learning*, um exemplo da utilização da técnica de amostragem é quando dividimos nosso *dataset* em dois conjuntos, um de treinamento (para construir o modelo, geralmente maior) e um de teste (para avaliar o modelo construído, geralmente menor). Essa operação se chama ***holdout***, e é importante porque, após construído, um modelo precisa ser validado com dados diferentes dos utilizados na sua construção. A figura a seguir ilustra o *holdout*:

Cliente	Ano	Mês	Dia	Atributos	Pagou?
João	2012	Janeiro	1	...	Sim
Oswaldo	2012	Janeiro	9	...	Sim
Maria	2012	Janeiro	21	...	Não
Roberta	2012	Fevereiro	4	...	Sim
...	...	...	...	...	...
Thiago	2013	Novembro	29	...	Sim
Dalva	2013	Dezembro	2	...	Não
Josy	2013	Dezembro	11	...	Não
Érica	2014	Janeiro	11	...	Sim
Renata	2014	Janeiro	19	...	Sim
...	...	...	...	...	...
Dorival	2014	Dezembro	20	...	Sim

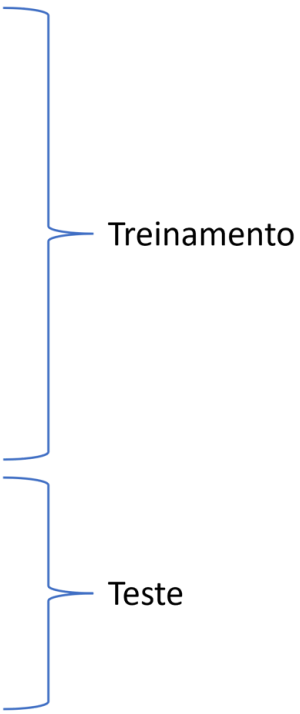


Figura 9.12: Holdout.

Outro exemplo de utilização da técnica de amostragem é a **validação cruzada** (*cross validation*), que pode ser estratificada ou não, e também é utilizada para avaliar modelos de *Machine Learning*. A validação cruzada consiste em dividir aleatoriamente o conjunto de dados com n elementos em k subconjuntos disjuntos com aproximadamente o mesmo número de elementos (ou seja, cada um com aproximadamente n/k). A cada rodada, cada um dos k subconjuntos é usado como conjunto de teste e os restantes são reunidos em um único conjunto de treino. O processo é repetido k vezes, sendo então gerados e avaliados k modelos, e essas avaliações são geralmente consolidadas através da média. Os valores mais comuns de k são 3, 5 e 10. A validação cruzada estratificada é similar, mas ao gerar os subconjuntos, a proporção de exemplos em cada uma das classes é considerada durante a amostragem.

Vamos exemplificar como seria a validação cruzada com *k* igual a 3, quando chamamos de validação cruzada *3-fold*. O primeiro passo é embaralhar os índices, como ilustra a figura a seguir:

Índice	Nome	...
1	Adriano	...
2	Mateus	...
3	Thiago	...
4	João	...
5	Pedro	...
6	Lucas	...
7	Paulo	...
8	Timóteo	...
9	Maria	...
10	Martha	...
11	Madalena	...
12	Ester	...

Índice	Nome	...
9	Maria	...
5	Pedro	...
6	Lucas	...
8	Timóteo	...
12	Ester	...
1	Adriano	...
10	Martha	...
2	Mateus	...
7	Paulo	...
11	Madalena	...
3	Thiago	...
4	João	...

Figura 9.13: Validação cruzada – Etapa 1.

Em seguida, são formados os 3 conjuntos, sendo que cada um deles será utilizado como conjunto de teste apenas uma vez, como ilustra a figura a seguir:

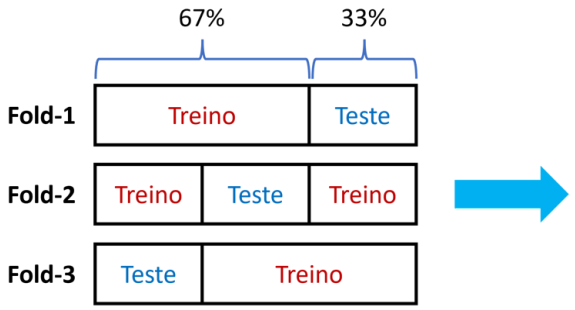
Índice	Nome	...	Pasta 1
9	Maria	...	Treino
5	Pedro	...	Treino
6	Lucas	...	Treino
8	Timóteo	...	Treino
12	Ester	...	Treino
1	Adriano	...	Treino
10	Martha	...	Treino
2	Mateus	...	Treino
7	Paulo	...	Teste
11	Madalena	...	Teste
3	Thiago	...	Teste
4	João	...	Teste

Índice	Nome	...	Pasta 2
9	Maria	...	Treino
5	Pedro	...	Treino
6	Lucas	...	Treino
8	Timóteo	...	Treino
12	Ester	...	Teste
1	Adriano	...	Teste
10	Martha	...	Teste
2	Mateus	...	Teste
7	Paulo	...	Treino
11	Madalena	...	Treino
3	Thiago	...	Treino
4	João	...	Treino

Índice	Nome	...	Pasta 3
9	Maria	...	Teste
5	Pedro	...	Teste
6	Lucas	...	Teste
8	Timóteo	...	Teste
12	Ester	...	Treino
1	Adriano	...	Treino
10	Martha	...	Treino
2	Mateus	...	Treino
7	Paulo	...	Treino
11	Madalena	...	Treino
3	Thiago	...	Treino
4	João	...	Treino

Figura 9.14: Validação cruzada – Etapa 2.

Assim, a acurácia de teste do modelo pode ser estimada calculando a média das acurácias de teste de cada *fold*, como ilustra a figura a seguir. A validação cruzada garante uma melhor estimativa da acurácia de teste do modelo, pois em vez de ter a acurácia de teste medida com apenas uma parte do *dataset* (como no *holdout*, que utiliza apenas o conjunto de teste), o modelo é testado, de alguma forma, com todos os dados do *dataset*, o que resulta em um melhor aproveitamento do *dataset*.



Fold	Acurácia	Exemplo
1	$AccTestePasta_1$	0,65
2	$AccTestePasta_2$	0,70
3	$AccTestePasta_3$	0,75
Final	*AccTeste*	(0,65 + 0,70 + 0,75)/3 = 0,70

Figura 9.15: Validação cruzada – Etapa 3.

Vale a pena ressaltar que principalmente quando temos um *dataset* desbalanceado (por exemplo, com 90% dos dados pertencendo à classe 1 e 10% pertencendo à classe 2), é essencial que utilizemos a **validação cruzada estratificada**. Para ilustrar esse ponto, observe o *dataset* desbalanceado da figura a seguir:

Índice	Nome	...	Peso (kg)	Altura (cm)	Sexo
1	Adriano	...	80	185	M
2	Mateus	...	67	175	M
3	Thiago	...	58	187	M
4	João	...	82	176	M
5	Pedro	...	74	168	M
6	Lucas	...	78	174	M
7	Paulo	...	71	172	M
8	Timóteo	...	92	179	M
9	Maria	...	54	167	F
10	Martha	...	58	170	F
11	Madalena	...	62	171	F
12	Ester	...	57	164	F

8 Homens = 67% da Base de Dados

4 Mulheres = 33% da Base de Dados

Figura 9.16: Validação cruzada estratificada – Etapa 1.

Neste caso, para executar a validação cruzada estratificada, o primeiro passo é embaralhar os índices de forma estratificada:

Índice	Nome	...
1	Adriano	...
2	Mateus	...
3	Thiago	...
4	João	...
5	Pedro	...
6	Lucas	...
7	Paulo	...
8	Timóteo	...
9	Maria	...
10	Martha	...
11	Madalena	...
12	Ester	...

Índice	Nome	...
5	Pedro	...
4	João	...
8	Timóteo	...
3	Thiago	...
6	Lucas	...
7	Paulo	...
1	Adriano	...
2	Mateus	...
11	Madalena	...
9	Maria	...
12	Ester	...
10	Martha	...

Figura 9.17: Validação cruzada estratificada – Etapa 2.

Em seguida, formamos os 3 conjuntos. Repare que a proporção de homens e mulheres é proporcional nos conjuntos de treino e teste:

Índice	Nome	...	Pasta 1
5	Pedro	...	Treino
4	João	...	Treino
8	Timóteo	...	Treino
3	Thiago	...	Treino
6	Lucas	...	Treino
7	Paulo	...	Treino
1	Adriano	...	Treino
2	Mateus	...	Treino
11	Madalena	...	Teste
9	Maria	...	Teste
12	Ester	...	Teste
10	Martha	...	Teste

Índice	Nome	...	Pasta 2
5	Pedro	...	Treino
4	João	...	Treino
8	Timóteo	...	Treino
3	Thiago	...	Treino
6	Lucas	...	Teste
7	Paulo	...	Teste
1	Adriano	...	Teste
2	Mateus	...	Teste
11	Madalena	...	Treino
9	Maria	...	Treino
12	Ester	...	Treino
10	Martha	...	Treino

Índice	Nome	...	Pasta 3
5	Pedro	...	Teste
4	João	...	Teste
8	Timóteo	...	Teste
3	Thiago	...	Teste
6	Lucas	...	Treino
7	Paulo	...	Treino
1	Adriano	...	Treino
2	Mateus	...	Treino
11	Madalena	...	Treino
9	Maria	...	Treino
12	Ester	...	Treino
10	Martha	...	Treino

Figura 9.18: Validação Cruzada Estratificada – Etapa 3.

Redução de dimensionalidade

O número de características de um *dataset* pode ser considerado a dimensionalidade dos dados. Por exemplo, 2 variáveis de entrada (colunas) delimitam um espaço com 2 dimensões, e cada exemplo (linha do *dataset*) é um ponto deste espaço, como ilustra a figura a seguir:

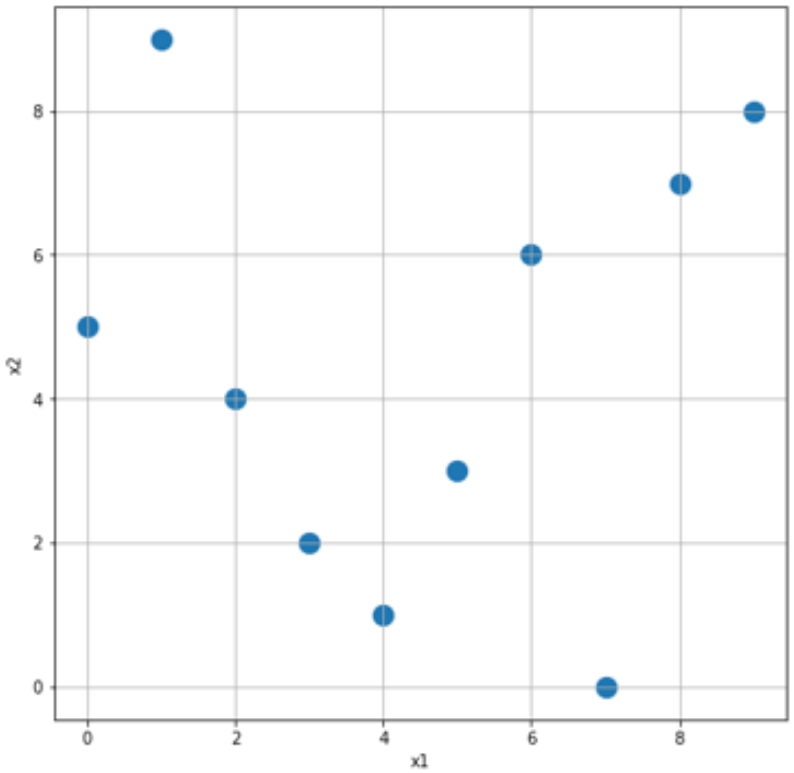


Figura 9.19: 2 dimensões.

Esse raciocínio pode ser escalado para qualquer número de variáveis de entrada para criar espaços multidimensionais. Entretanto, quanto maior o número de dimensões de um espaço, maior é o volume deste espaço, sendo provável que esse *dataset* represente uma amostragem muito esparsa e provavelmente não representativa desse espaço. Isso é conhecido como **maldição da dimensionalidade**, e é ilustrado pela figura a seguir:

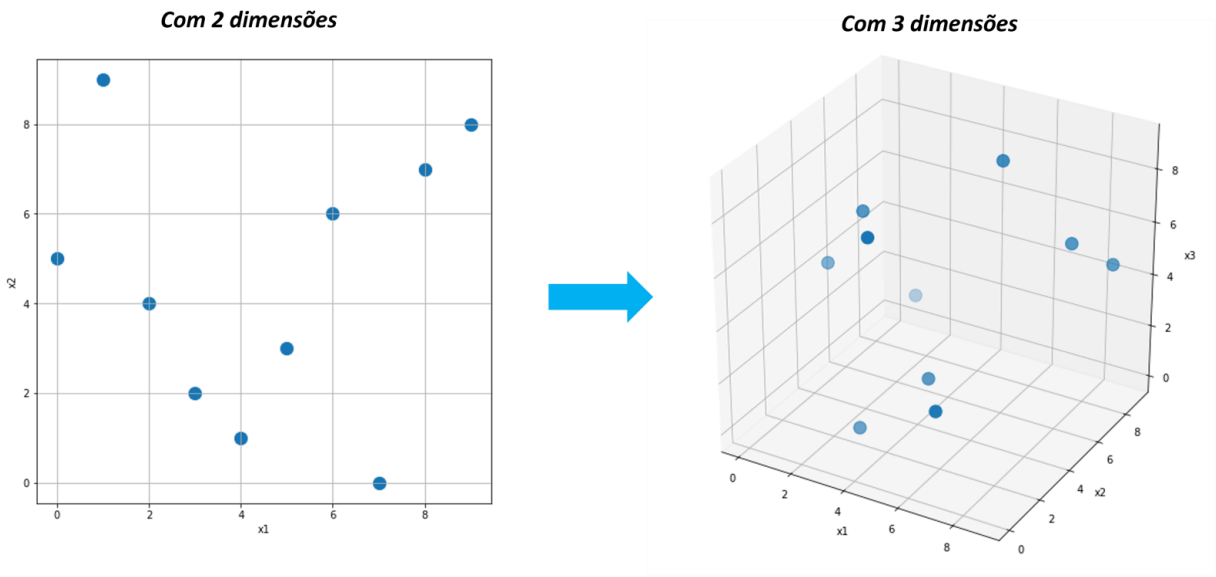


Figura 9.20: 2 e 3 dimensões.

Para atuar nesse problema, a redução de dimensionalidade consiste em projetar os dados em um espaço de dimensão inferior, mas que ainda preserva as propriedades mais importantes do *dataset* original. No mundo real, os *datasets* podem ter um grande número de características, e muitos algoritmos de *Machine Learning* funcionam melhor se a dimensionalidade for menor.

São exemplos da técnica de redução de dimensionalidade a **análise de componentes principais (PCA)**, uma técnica da álgebra linear que encontra novos atributos (componentes principais) que sejam combinações lineares dos originais; e as técnicas de ***feature selection* (seleção de características)**, que visam remover colunas não relevantes ou redundantes, e que detalharemos a seguir. Adicionalmente, a **agregação**, que vimos anteriormente, também pode ser considerada uma técnica de redução de dimensionalidade, pois reduz o número de colunas do *dataset*.

Como vantagens da aplicação da técnica de redução de dimensionalidade, podemos citar que um *dataset* menor tem menor custo de processamento e que a redução de dimensionalidade pode eliminar características irrelevantes, reduzir ruídos e remover

dependências lineares entre os atributos (como variáveis correlacionadas). Entretanto, como desvantagem, caso sejam aplicadas técnicas como o PCA, as "novas" variáveis não estarão diretamente relacionadas às variáveis de entrada originais, tornando a projeção difícil de interpretar.

Seleção de características

Também conhecida como *feature selection*, a seleção de características consiste em usar apenas um subconjunto das características (atributos) originais, desconsiderando as características redundantes ou irrelevantes. Para isso, existem diversas abordagens: pode ser utilizado um algoritmo que decide quais atributos usar e quais ignorar; os atributos podem ser pré-selecionados por um especialista ou pode ser executado um algoritmo que encontre o melhor subconjunto de atributos (geralmente aqueles mais relacionados à variável de saída). A figura a seguir ilustra a seleção de características:

Id	Nome	Sexo	Idade
123	Tatiana	F	16
456	Rubens	M	50
780	Adriano	M	28
990	João	M	35

Sexo	Idade
F	16
M	50
M	28
M	35

Figura 9.21: Seleção de características.

Criação de recursos

Também conhecida como *feature engineering*, a criação de recursos consiste em criar, a partir dos atributos originais, um novo conjunto de atributos que capture informações importantes. Para isso, podem ser utilizadas diversas técnicas, por exemplo:

- Criar uma nova coluna booleana para indicar a presença ou ausência de borda em uma foto;
- Criar uma característica agrupada ou resumida, como o IMC, que usa as características peso e altura;

- Agregar mais informações a cada registro do conjunto de dados, com pesquisas, consultas a bases de dados externas ou coleta de mais dados.

Transformação de dados

São exemplos da técnica de transformação de dados:

Normalização

A **normalização** redimensiona os dados de um ou mais atributos do intervalo original para o intervalo de 0 a 1, com o objetivo de alterar os valores para uma escala comum, sem distorcer as diferenças nos intervalos de valores. É uma boa técnica para usar quando não se conhece bem a distribuição dos dados ou quando se sabe que a distribuição não é *gaussiana* (*normal*).

São utilizados os valores mínimo e máximo observáveis, sendo possível estimar esses valores a partir dos dados disponíveis. Normalizamos os dados dividindo todos os valores pelo valor máximo encontrado ou subtraindo o valor mínimo e dividindo pelo intervalo entre os valores máximo e mínimo:

```
y = (x – min) / (max – min)
```

Vejamos um exemplo de normalização na tabela a seguir:

Original	Normalizado
100	1
8	0.042
50	0.479
88	0.875
4	0

Padronização

A **padronização** redimensiona a distribuição de um ou mais atributos dos valores observados para que eles tenham as propriedades de uma distribuição normal padrão (média 0 e desvio padrão 1). A padronização pressupõe que suas observações sigam uma distribuição *normal*. Ainda é possível padronizar seus dados mesmo se essa premissa não for verdadeira, mas a técnica é mais eficaz se os dados seguirem essa distribuição.

Para padronizar os dados, calcula-se a média estatística e o desvio padrão dos valores dos atributos, subtrai-se a média de cada valor e divide-se o resultado pelo desvio padrão. Um valor é padronizado da seguinte maneira:

```
média = soma (x) / contagem (x)

desvio padrão = $\sqrt{(soma ((x - média)^2) / contagem (x))}$

y = (x - média) / desvio padrão
```

Vejamos um exemplo de padronização na tabela a seguir:

Original	Padronizado
100	1.264
8	-1.062
50	0
88	0.961
4	-1.163

Quando normalizar e quando padronizar?

- Se a distribuição é normal, padronize. Caso contrário, normalize.
- Os problemas de modelagem preditiva são muitas vezes complexos, não sendo clara a melhor transformação para realizar. Na dúvida, use a normalização. Se tiver tempo, explore os modelos com os dados sem transformação, com a padronização e com a normalização e veja se os resultados são significativamente diferentes e se o custo-benefício vale a pena.
- Como a padronização resulta em valores positivos e negativos, pode ser interessante normalizar os dados após a padronização.
- É possível definir os valores de mínimo e máximo de acordo com o conhecimento no negócio (e não simplesmente se ater aos valores observados).

Discretização

A **discretização** consiste em transformar um atributo contínuo em categorias ordinais, como ilustra a tabela a seguir:

Salário	Valor discretizado
R$ 1.000,00	Baixo
R$ 800,00	Baixo
R$ 5.000,00	Médio
R$ 10.000,00	Alto
R$ 20.000,00	Alto

One-hot encoding e *Dummy variable encoding*

Consiste em transformar atributos categóricos nominais (ou seja, que não haja ordenação existente entre eles) em "colunas binárias". Para isso, é criada uma variável binária para cada valor único da variável original.

O *one-hot encoding* cria uma variável binária para cada categoria existente, mas essa representação inclui redundância. Por exemplo, se soubermos que [1, 0, 0] representa "azuis" e [0, 1, 0] representa "verdes", não precisamos de outra variável binária [0, 0, 1] para representar "castanhos". Poderíamos usar [0, 0] para "castanhos", [1, 0] para "azuis" e [0, 1] para "verdes", como na codificação de variável *dummy*, que representa *c* categorias em *c-1* variáveis binárias. As duas técnicas são ilustradas pela figura a seguir:

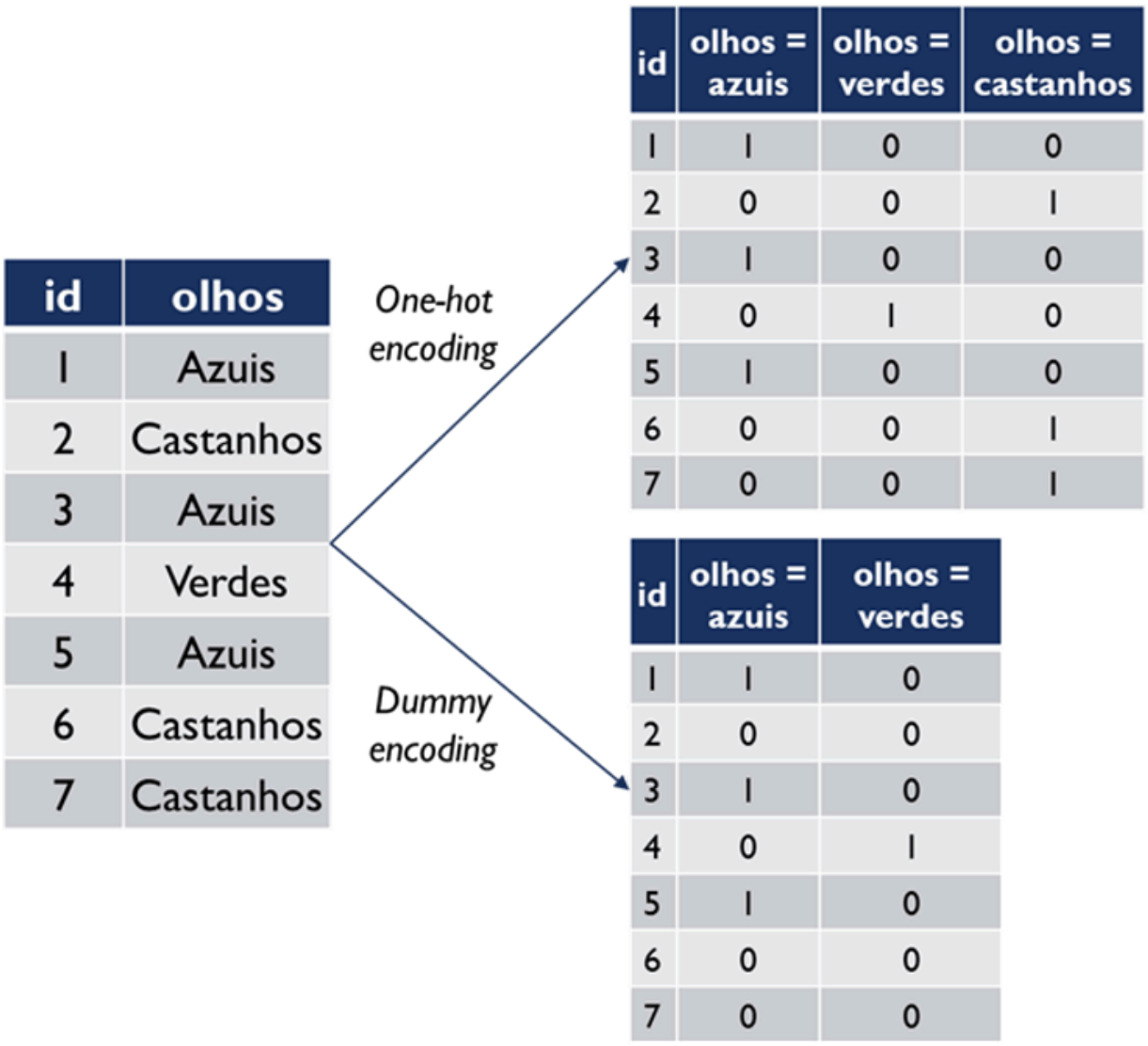


Figura 9.22: One-hot encoding / Dummy variable encoding.

Atenção! Cuidado com o *Data Leakage*!

No caso da *normalização* e da *padronização*, uma abordagem ingênua (e incorreta) para preparar dados aplica a transformação em todo o conjunto de dados (antes do particionamento em bases de treino e teste). Isso resulta em um problema conhecido como **vazamento de dados** (*data leakage*): algum conhecimento sobre o conjunto de teste vaza para o conjunto de dados usado para treinar o modelo, podendo resultar em uma estimativa incorreta do desempenho do modelo ao fazer predições sobre novos dados.

Por exemplo, quando fazemos a normalização, convertemos as variáveis de entrada para um intervalo entre 0 e 1. Para isso, precisamos calcular os valores mínimo e máximo para cada variável e então reescalar as variáveis. Porém, se fizermos primeiro isso com todo o *dataset* e depois o particionarmos em conjuntos de treino e teste, possivelmente o conjunto de treino saberá algo sobre o conjunto de teste, pois foram utilizados os valores mínimo e máximo globais, e não apenas os do conjunto de treino.

A forma correta de aplicar essas transformações é:

- Dividir o *dataset* em conjuntos de treino e teste;
- Ajustar a preparação de dados no conjunto de dados de treino (por exemplo, determinar o mínimo e o máximo deste conjunto);
- Aplicar a preparação de dados no conjunto de treino e teste separadamente (utilizando os valores mínimo e máximo do conjunto de treino);
- Construir e avaliar os modelos.

9.2 Exemplo prático

Você pode acessar o código-fonte do exemplo prático de pré-processamento de dados em Python pelo seguinte link: https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_pre_processamento.ipynb

Para exemplificar como funciona a etapa de pré-processamento de dados, vamos utilizar novamente o *dataset Iris* e ilustrar a limpeza de dados, em especial, o tratamento de *missings*. Em seguida, vamos utilizar alguns *datasets* artificiais para falarmos sobre as operações de transformação de dados, como a normalização, a padronização, *One-hot encoding* e *Dummy variable encoding*.

Iniciaremos esta prática importando os pacotes que utilizaremos:

```
# Importação de pacotes
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler # normalização
from sklearn.preprocessing import StandardScaler # padronização
from sklearn.preprocessing import OrdinalEncoder # ordinal encoding
from sklearn.preprocessing import OneHotEncoder # one-hot encoding e dummy
variable encoding
```

Agora vamos importar o *dataset Iris* a partir de um link e armazenar os dados em um *dataframe*, da mesma forma que fizemos na prática anterior. Exibiremos também as 5 últimas linhas do dataframe:

```
# importando dados de uma url para um dataframe

# url a importar
url_dados =
'https://raw.githubusercontent.com/profkalinowski/livroescd/main/iris.data
'


# labels dos atributos do dataset
labels_atributos = ['comprimento_sepala', 'largura_sepala',
'comprimento_petala', 'largura_petala', 'especie']

# carga do dataset através da url
iris = pd.read_csv(url_dados, names=labels_atributos)

# exibindo as últimas linhas
iris.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figura 9.23: Saída.

Como vimos na prática anterior, o *dataset Iris* não possui valores faltantes (*missings*). Agora, vamos criar uma cópia desse *dataframe*, adicionando uma nova linha que contém um valor *missing*. Exibiremos novamente as 5 últimas linhas para verificar a nova linha que inserimos:

```
# criando um novo dataframe com iris + uma nova linha com um valor missing
para o próximo exemplo
df = iris.append({'largura_sepala': 4.0,
                  'comprimento_petala': 5.0,
                  'largura_petala': 0.4,
                  'especie': 'Iris-setosa'},
                  ignore_index=True)
df.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
150	NaN	4.0	5.0	0.4	Iris-setosa

Figura 9.24: Saída.

Em seguida, vamos executar um comando para eliminar deste *dataframe* as linhas que tiverem qualquer valor *missing*. Nesse caso, será a última linha que adicionamos:

```
# eliminando linhas que tenham ALGUM valor missing

df = df.dropna(how='any')
df.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figura 9.25: Saída.

Vejamos um novo exemplo de tratamento de *missings*. Vamos criar novamente um novo *dataframe* com a cópia do *dataframe* original. Em seguida, adicionamos uma nova linha com um valor *missing*, e uma nova linha com todos os valores *missing*:

```
# criando (novamente) um novo dataframe com iris + uma nova linha com um
valor missing para o próximo exemplo
df = iris.append({'largura_sepala': 4.0,
                  'comprimento_petala': 5.0,
                  'largura_petala': 0.4,
                  'especie': 'Iris-setosa'},
                  ignore_index=True)

# adicionando uma nova linha com todos os valores missing para o próximo
exemplo
df = df.append({}, ignore_index=True)
df.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
150	NaN	4.0	5.0	0.4	Iris-setosa
151	NaN	NaN	NaN	NaN	NaN

Figura 9.26: Saída.

Vamos eliminar deste *dataframe* as linhas que tiverem todos os valores *missing*. Neste caso, será removida a última linha que adicionamos, mas não a anterior:

```
# eliminando linhas que tenham TODOS os valores missing
df = df.dropna(how='all')
df.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
150	NaN	4.0	5.0	0.4	Iris-setosa

Figura 9.27: Saída

Em seguida, definimos valores para o preenchimento de *missings*, e realizamos o preenchimento dos *missings* com estes valores definidos:

```
# definindo valores para o preenchimento de missings
values = {'comprimento_sepala': iris['comprimento_sepala'].median(),
          'largura_sepala': 5.0,
          'comprimento_petala': 4.0,
```

```
          'largura_petala': 0.1,
          'especie': 'Iris-setosa'}

# fazendo o preenchimento
df = df.fillna(value=values)
df.tail()
```

	comprimento_sepala	largura_sepala	comprimento_petala	largura_petala	especie
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
150	5.8	4.0	5.0	0.4	Iris-setosa

Figura 9.28: Saída.

Agora que já vimos como fazer o tratamento de *missings*, vamos entender como fazer transformações numéricas e categóricas.

Transformações numéricas

Podemos utilizar as operações de normalização e padronização usando o módulo de pré-processamento da biblioteca *Scikit-learn* (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing). Para normalizar os dados, usamos o *MinMaxScaler* e para padronizar os dados, usamos o *StandardScaler*.

Para exemplificar as transformações numéricas, utilizaremos um *dataset* pequeno e artificial, para que seja fácil notar as transformações realizadas:

```
# dados que vamos usar nos exemplos
data = np.asarray([[100, 0.001],
                   [8,   0.05],
                   [50,  0.005],
                   [88,  0.07],
```

```
                  [4,   0.1]])
print(data)
```

Saída:

```
[[1.0e+02 1.0e-03]
 [8.0e+00 5.0e-02]
 [5.0e+01 5.0e-03]
 [8.8e+01 7.0e-02]
 [4.0e+00 1.0e-01]]
```

Vamos normalizar os dados originais e, em seguida, padronizá-los. Os comentários nos blocos de código auxiliam no seu entendimento.

Normalização

```
# Normalização

# definindo o transformador como min max scaler
scaler = MinMaxScaler()

# transformando os dados
scaled = scaler.fit_transform(data)
print(scaled)
```

Saída:

```
[[1.         0.        ]
 [0.04166667 0.49494949]
 [0.47916667 0.04040404]
 [0.875      0.6969697 ]
 [0.         1.        ]]
```

Padronização

```
# Padronização

# definindo o transformador como standard scaler
scaler = StandardScaler()

# transformando os dados
```

```
scaled = scaler.fit_transform(data)
print(scaled)
```

Saída:

```
[[ 1.26398112 -1.16389967]
 [-1.06174414  0.12639634]
 [ 0.         -1.05856939]
 [ 0.96062565  0.65304778]
 [-1.16286263  1.44302493]]
```

Transformações categóricas

Podemos utilizar as operações *one-hot encoding* e *dummy variable encoding* usando o módulo de pré-processamento da biblioteca *Scikit-learn* (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing). Para o *one-hot encoding*, usamos o *OneHotEncoder* e para o *dummy variable encoding*, também usamos o *OneHotEncoder*, mas usando o parâmetro *drop* para indicar qual categoria receberá todos os valores zero.

O *OneHotEncoder* ordena as categorias alfabeticamente antes de aplicar a transformação. É possível especificar a lista de categorias através do parâmetro *categories*. Espera-se que o conjunto de treinamento contenha pelo menos um exemplo de cada categoria se as categorias não forem explicitamente definidas. Se os novos dados (conjunto de teste, por exemplo) tiverem categorias não vistas no treinamento, é possível configurar o parâmetro *handle_unknown* como *ignore* para que não ocorra um erro.

Para exemplificar as transformações categóricas, utilizaremos um *dataset* pequeno e artificial, para que seja fácil notar as transformações realizadas:

```
# dados que vamos usar nos exemplos
data = np.asarray([['castanhos'], ['verdes'], ['azuis'], ['azuis'],
['castanhos'], ['azuis']])
print(data)
```

Saída:

```
[['castanhos']
 ['verdes']
 ['azuis']
 ['azuis']
 ['castanhos']
 ['azuis']]
```

A seguir, vamos aplicar o *one-hot encoding* nos dados originais e, em seguida, aplicar o *dummy variable encoding* também nos dados originais. Os comentários nos blocos de código auxiliam no seu entendimento.

One-Hot Encoding

```
# definindo o transformador como one-hot encoding
encoder = OneHotEncoder(sparse=False)

# transformando os dados
onehot = encoder.fit_transform(data)
print(onehot)
```

Saída:

```
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [1. 0. 0.]]
```

Dummy Variable Encoding

```
# definindo o transformador como one-hot encoding (com dummy variable
encoder)
encoder = OneHotEncoder(drop='first', sparse=False)
```

```
# transformando os dados
dummy = encoder.fit_transform(data)
print(dummy)
```

Saída:

```
[[1. 0.]
 [0. 1.]
 [0. 0.]
 [0. 0.]
 [1. 0.]
 [0. 0.]]
```

É importante ressaltar que, neste capítulo, para fins de exemplificação, mostramos apenas algumas das possíveis operações de pré-processamento de dados dentre as muitas existentes. É necessário sempre entender bem os seus dados para então decidir quais os tratamentos mais adequados em cada situação.

CAPÍTULO 10
Algoritmos de Machine Learning para classificação e regressão

No contexto de Ciência de Dados, muitas vezes trabalhamos com modelos de *Machine Learning* a fim de construirmos modelos estatísticos capazes de "aprender" com os nossos dados e fazer predições. Este aprendizado pode ser categorizado em quatro tipos:

- **supervisionado**, quando o modelo de conhecimento é construído a partir dos dados apresentados na forma de pares ordenados (entrada, saída desejada);
- **não supervisionado**, quando não existe a informação da saída desejada e o processo de aprendizado busca identificar regularidades entre os dados a fim de agrupá-los em função das similaridades que apresentam entre si;
- **semi-supervisionado**, que combina o aprendizado supervisionado e o não supervisionado; e
- **aprendizado por reforço**, quando a máquina é capaz de "perceber" o estado do ambiente, executar ações de acordo com esse estado, receber "recompensas" de acordo com as ações executadas e trocar de estado, se apropriado.

Neste capítulo, vamos focar no **aprendizado supervisionado**, que trata dos problemas de *classificação* e *regressão*.

Quando temos uma pergunta como "*Devo conceder ou não crédito para um cliente?*", temos um problema de ***classificação***, pois a variável a ser predita é categórica, representada pelas classes "sim" e "não". Já se a pergunta fosse "*Devo conceder qual valor de crédito para um cliente?*", teríamos um problema de ***regressão***, pois desejamos fazer a predição de um valor numérico, contínuo ou discreto. Para ambos os casos, podemos aprender com dados passados de clientes que fizeram empréstimos para decidir o que fazer quando chegar a solicitação de um novo cliente.

Relembrando: no **aprendizado supervisionado**, o modelo é construído a partir dos dados de entrada (também chamados de *dataset*), que são apresentados para um algoritmo na forma de pares ordenados **entrada – saída desejada**. Dizemos que estes dados são rotulados, pois sabemos de antemão a saída esperada para cada entrada de dados. Neste caso, o aprendizado (ou treinamento) consiste em apresentarmos para o algoritmo um número suficiente de exemplos (também chamados de registros ou instâncias) de entradas e saídas desejadas já rotuladas previamente.

O objetivo do algoritmo é aprender uma regra geral que mapeie as entradas nas saídas corretamente, o que consiste no modelo final. Os dados de entrada podem ser divididos em dois grupos:

- *X*, com os *atributos* (também chamados de *características*) a serem utilizados na determinação da classe de saída (também chamados de atributos *previsores* ou *de predição*);
- *Y*, com o atributo para o qual se deseja fazer a predição do valor de saída categórico ou numérico (também chamado de *atributo-alvo*, ou *target*).

É comum que particionemos os dados de entrada rotulados em dois conjuntos: o conjunto de **treinamento**, que servirá para construir o modelo, e o conjunto de **teste** (também chamado na literatura de conjunto de validação), que servirá para verificar como o modelo se comportaria em dados não vistos, de forma que possamos ajustá-lo, se necessário, para a construção final do modelo a ser aplicado em novos dados cuja saída esperada ainda não conhecemos.

Lembrando do esquema básico de um projeto de Ciência de Dados, apresentado anteriormente neste livro, neste capítulo vamos focar na etapa de **modelagem e inferência**:

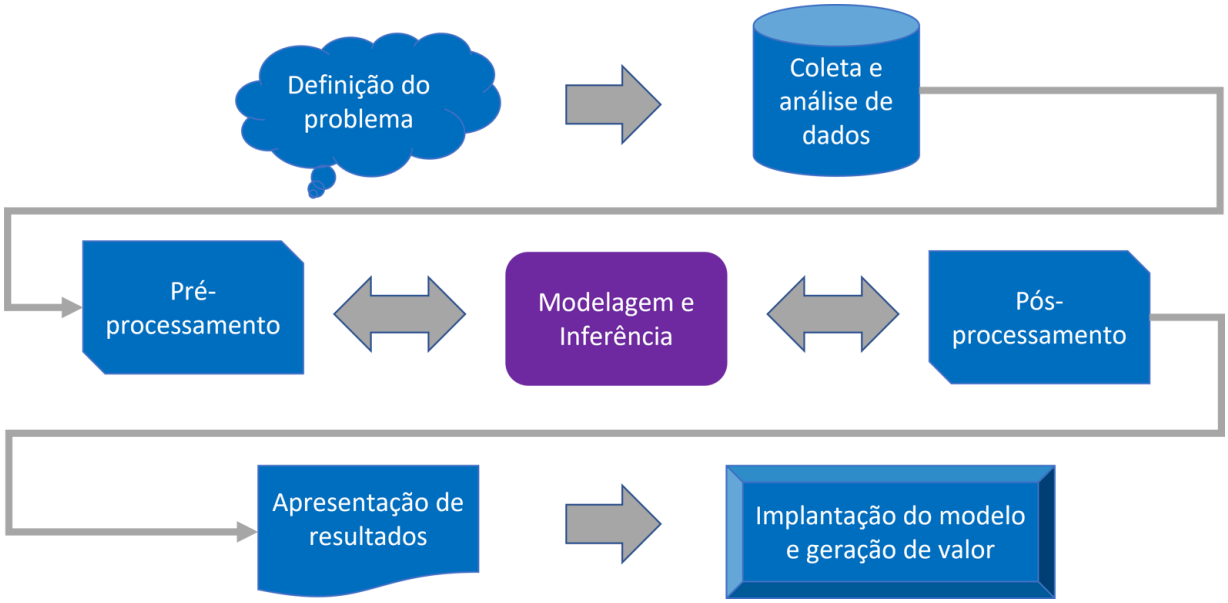


Figura 10.1: Esquema básico de um projeto de Ciência de Dados – Etapa de modelagem e inferência.

Dentro do ciclo de vida de projetos de Ciência de Dados, a etapa de **modelagem e inferência** consiste, de forma simplificada, em escolher o modelo mais adequado para resolver o problema em questão. Resumidamente, esta etapa consiste em três tarefas:

- Elencar os modelos possíveis e passíveis para cada tipo de problema;
- Estimar os parâmetros que compõem os modelos, baseando-se nas instâncias e variáveis pré-processadas;
- Avaliar os resultados de cada modelo, usando métricas e um processo justo de comparação.

Em problemas de **classificação**, a saída é sempre categórica (classe) e em problemas de **regressão**, numérica (discreta ou contínua), e para ambos os problemas, dado um novo padrão, espera-se que o modelo estime a classe ou o valor mais esperado para a variável resposta. Veremos mais detalhes sobre os problemas de classificação e regressão a seguir.

10.1 Problemas de classificação

A **classificação** é uma das tarefas de *Machine Learning* mais importantes e mais populares. Conforme já mencionado, problemas de classificação utilizam o aprendizado supervisionado: apresentamos, para o algoritmo, um número suficiente de exemplos já classificados previamente compostos por entradas e suas saídas correspondentes, e o objetivo do algoritmo é aprender uma regra geral que mapeie as entradas nas saídas corretamente. Vimos também que os dados de entrada podem ser divididos em dois grupos: *X*, com os atributos a serem utilizados na determinação da classe de saída, e *Y*, com o atributo para o qual se deseja fazer a predição do valor da classe.

Podemos definir um problema de classificação informalmente como a busca por uma função matemática que permita associar corretamente cada exemplo X_i de um conjunto de dados a um único rótulo categórico Y_i, denominado *classe*. Uma vez identificada, essa função pode ser aplicada a novos exemplos para prever as classes em que eles se enquadram.

Formalmente, podemos definir um problema de classificação como: dada uma coleção de exemplos *f*, obter uma função (hipótese) *h* que seja uma aproximação de *f*. A imagem de *f* é formada por rótulos de classes retirados de um conjunto finito, e toda hipótese *h* é chamada de *classificador*. O aprendizado consiste, então, na busca da hipótese *h* que mais se aproxime da função original *f*.

O fluxo de um problema de classificação pode ser representado da seguinte forma: a partir de uma base de dados rotulada (para cada exemplo, conhecemos a sua respectiva classe), geram-se dois subconjuntos disjuntos — a **base de treino** (contendo, por exemplo, 70% dos dados originais) e a **base de teste** (contendo, por exemplo, 30% dos dados originais).

A base de treino é submetida ao classificador para treinamento do modelo, que é calibrado de acordo com os dados apresentados. Após essa etapa, apresentam-se os exemplos da base de teste para o modelo, que deverá realizar a predição de suas classes. Comparando-se as classes preditas com as classes verdadeiras da base de teste, pode-se medir a qualidade do modelo, isto é, sua habilidade em classificar corretamente exemplos não vistos durante o treinamento. Este fluxo pode ser resumido pela figura:

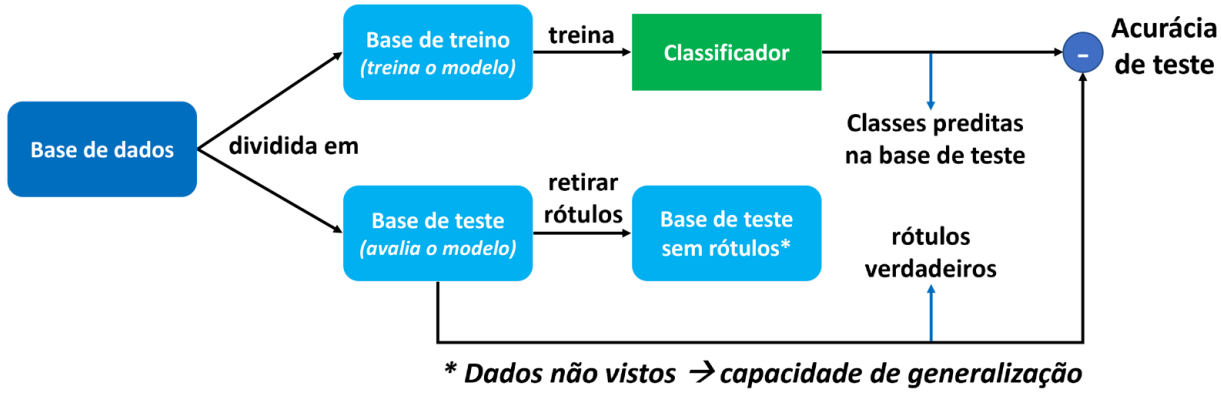


Figura 10.2: Fluxo resumido de um problema de classificação.

Quando construímos um modelo de aprendizado supervisionado, estamos interessados na generalização, etapa em que aplicamos um modelo aos dados que não foram utilizados para construí-lo, ou seja, novos dados. Podemos pensar que cada *dataset* é uma amostra finita de uma população e que os dados de treinamento também vieram dessa população. Queremos que os modelos sejam bons não apenas para o conjunto de treinamento (usado para construir o modelo), mas para a população em geral. Para isso, podemos usar a técnica de *holdout* (separação do *dataset* em bases de treino e teste): o modelo é construído com um conjunto de dados e seu erro de generalização é avaliado com outro (não utilizado para o treinamento).

Vale a pena observar que, uma vez identificada uma hipótese (uma vez treinado um classificador), esta pode ser muito específica para o conjunto de treino utilizado. Caso esse conjunto não corresponda a

uma amostra suficientemente representativa da população, esse classificador pode ter um bom desempenho no conjunto de treino, mas não no conjunto de teste. Dizemos que ocorreu ***overfitting*** quando o classificador se ajustou em excesso ao conjunto de treinamento.

Geralmente, o erro de teste é maior que o erro de treinamento. Idealmente, esses erros são próximos. O modelo tem boa capacidade de generalização, sendo capaz de aprender bem exemplos ainda não vistos no treinamento. Se o erro de teste é grande demais em relação ao erro de treino, isso é um indicativo de que pode estar ocorrendo *overfitting* e o modelo está memorizando os padrões de treinamento em vez de descobrir regras ou padrões generalizados. Em geral, modelos mais simples tendem a generalizar melhor. Um modelo bem ajustado apresentará erros da mesma magnitude tanto nos novos dados quanto nos dados de treino, enquanto um modelo com *overfitting* "memoriza" os dados de treino e apresentará erros muito maiores nos novos dados.

Outra situação que pode acontecer é se o algoritmo de aprendizado tiver parametrizações inadequadas. Neste caso, diz-se que ocorreu ***underfitting***: o classificador se ajustou pouco ao conjunto de treinamento, não sendo adequado para realizar predições no conjunto de teste. Para resolver esse problema, podemos experimentar um modelo mais complexo, que seja capaz de aprender os padrões do conjunto de treinamento de forma mais adequada.

Na aprendizagem supervisionada, ao mesmo tempo em que o modelo preditivo precisa ser suficientemente flexível para aproximar os dados de treinamento, o processo de treinamento deve evitar que o modelo absorva os ruídos da base. O modelo deve capturar as regularidades dos dados de treinamento, mas também generalizar bem para dados desconhecidos (conjunto de teste). A solução para isso é escolher o momento adequado para interromper o treinamento e/ou utilizar validação cruzada.

Chegamos, então, a um dilema: por um lado, queremos que nosso algoritmo modele muito bem os dados de treinamento, caso contrário, perderemos recursos relevantes e tendências interessantes. Por outro lado, não queremos que nosso modelo se encaixe muito perfeitamente aos dados de treinamento, arriscando aprender todos os erros e irregularidades. Tentando evitar o *overfitting*, podemos começar a ir para o outro extremo, fazendo com que nosso modelo comece a ignorar recursos importantes do nosso *dataset*. Isso acontece quando escolhemos um modelo que não é complexo o suficiente para capturar esses recursos importantes, ocorrendo *underfitting*.

Como dito anteriormente, o modelo deve capturar as regularidades dos dados de treinamento, mas também generalizar bem para dados desconhecidos. Isto é conhecido como **dilema *bias* x variância**, e fala sobre a importância do equilíbrio entre *underfitting* e *overfitting*.

Uma das formas de lidar com o dilema *bias* x variância é utilizar a validação cruzada (*cross validation*), técnica já detalhada no capítulo anterior. Relembrando, a validação cruzada estima o desempenho de generalização do modelo não apenas uma vez - como no *holdout* -, mas diversas vezes (*k*), fornecendo estatísticas sobre o desempenho estimado (como média e variância) para que possamos entender como é esperado que o desempenho varie entre os conjuntos de dados e termos mais confiança na estimativa de desempenho, fazendo um melhor uso de um conjunto limitado de dados.

Métricas de avaliação

Vale a pena mencionar o teorema "não existe almoço grátis": não existe um algoritmo de aprendizado que seja superior a todos os demais quando considerados todos os problemas possíveis. A cada problema, os algoritmos disponíveis devem ser experimentados, com diversas configurações, a fim de identificar aqueles que obtêm

melhor desempenho. Para isso, precisamos utilizar métricas de avaliação apropriadas.

Existem diversas medidas para estimar o desempenho de um modelo de aprendizado supervisionado, ou seja, avaliar o modelo. Para problemas de classificação, a **matriz de confusão** oferece um detalhamento do desempenho do modelo, mostrando, para cada classe, o número de classificações corretas em relação ao número de classificações indicadas pelo modelo, como ilustra a tabela a seguir:

Classes	Predita C1	Predita C2
Verdadeira C1	Verdadeiros Positivos (VP)	Falsos Negativos (FN)
Verdadeira C2	Falsos Positivos (FP)	Verdadeiros Negativos (VN)

Para problemas de classificação, uma das métricas mais utilizadas é a **acurácia**, ou taxa de acerto do classificador, dada por:

$$Acc(h) = 1 - Err(h)$$

Figura 10.3: Acurácia.

A acurácia é uma função da taxa de erro (ou taxa de classificação incorreta), dada por:

$$Err(h) = \frac{1}{n} \sum_{i=1}^{n} ||y_i \neq h(i)||$$

Figura 10.4: Erro.

Onde:

- O operador //*E*// retorna 1 se a expressão *E* for verdadeira e 0 em caso contrário;
- *n* é o número de exemplos (registros da base de dados);
- y_i é a classe real associada ao i-ésimo exemplo;
- *h(i)* é a classe indicada pelo classificador para o i-ésimo exemplo.

Esta é uma métrica útil quando os erros nas predições de todas as classes são igualmente importantes. A acurácia também pode ser extraída da matriz de confusão, pois representa o número de exemplos classificados corretamente dividido pelo número total de exemplos classificados:

$$\text{Acurácia} = \frac{VP+VN}{VP+VN+FP+FN}$$

Figura 10.5: Acurácia.

Também a partir da matriz de confusão temos a **precisão** e o ***recall***:

- A precisão é a razão entre o número de predições VP sobre a quantidade total de predições positivas.
- O recall é a razão entre o número de predições VP sobre a quantidade total de exemplos positivos no *dataset*.

$$\text{Precisão} = \frac{VP}{VP+FP}$$

$$\text{Recall} = \frac{VP}{VP+FN}$$

Figura 10.6: Precisão e recall.

Na prática, quase sempre temos que escolher entre uma alta precisão ou um alto *recall*. Tipicamente, é impossível ter ambos.

Outra métrica bastante utilizada em problemas de classificação binária é a **AUC (*Area under curve*)**, que varia entre 0 (predições 100% incorretas) e 1 (predições 100% corretas). Esta métrica é extraída da **curva ROC (*Receiver Operating Characteristic*)**, uma representação gráfica do desempenho de um classificador. Seu gráfico contrasta, em seus dois eixos, os benefícios de uma classificação correta (sensibilidade, *recall* ou taxa de verdadeiros positivos - TVP) e o custo de uma classificação incorreta (*1-especificidade*, ou *Taxa de Falsos Positivos* - TFP), sendo:

- **Sensibilidade**: capacidade de identificar corretamente os indivíduos que apresentam a característica de interesse. É similar ao *recall*, que vimos anteriormente.
- **Especificidade**: capacidade de identificar corretamente os indivíduos que não apresentam a condição de interesse.

$$\text{Sensibilidade} = \text{TVP} = \frac{VP}{VP+FN}$$

$$\text{Especificidade} = \frac{VN}{FP+VN}$$

$$1-\text{Especificidade} = \text{TFP} = \frac{FP}{FP+VN}$$

Figura 10.7: Sensibilidade e especificidade.

Assim, a curva ROC é produzida calculando e plotando em um gráfico a taxa de verdadeiros positivos em relação à taxa de falsos positivos para um único classificador em vários limites e, a partir dela, é calculada a AUC, como ilustra a figura a seguir.

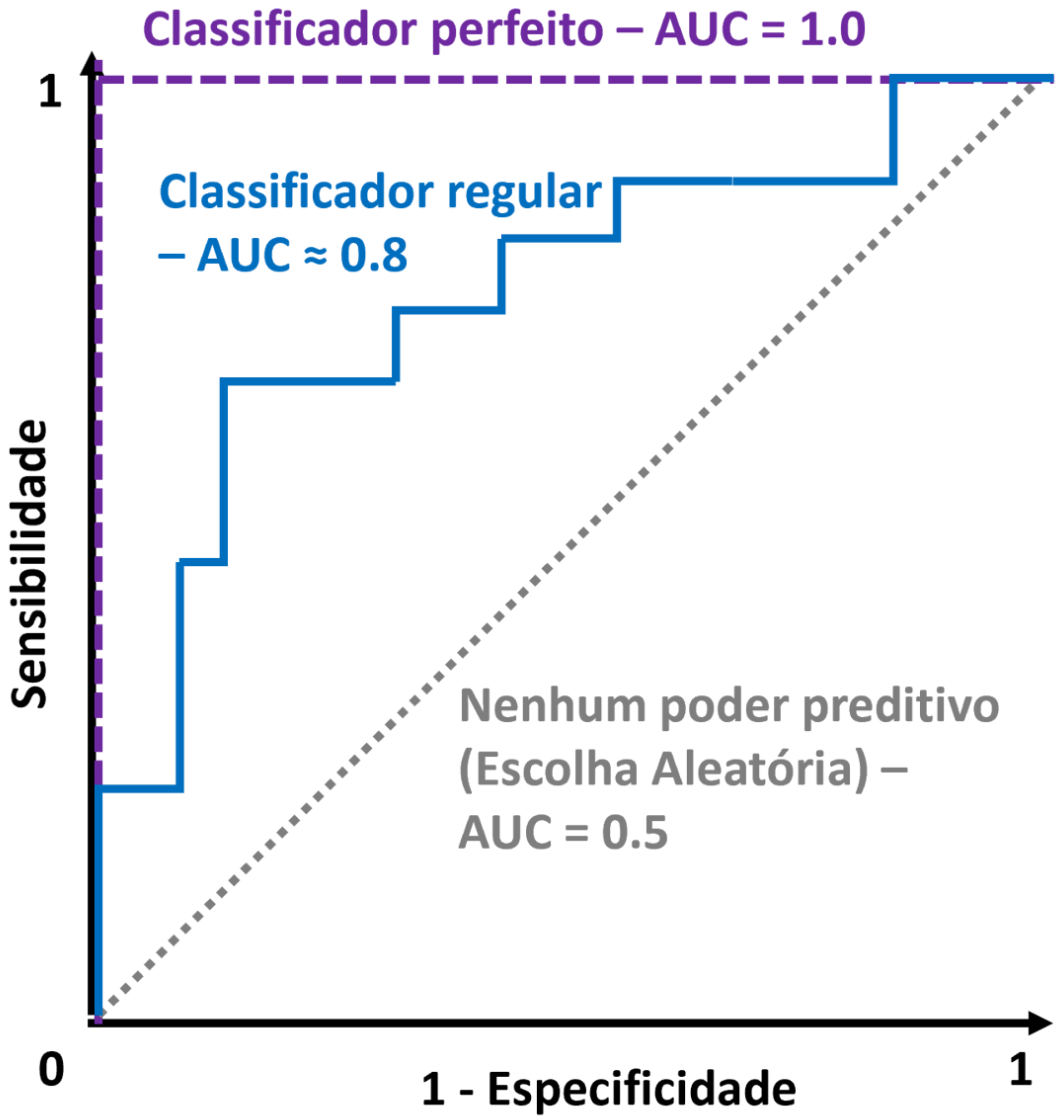


Figura 10.8: Exemplo de curvas ROC e suas respectivas AUCs.

10.2 Problemas de regressão

Os problemas de classificação podem ser considerados um subtipo dos **problemas de regressão** (também conhecidos como problemas *de estimação*), pois funcionam de forma similar, com a diferença de que, em vez de o resultado ser categórico, como é na

classificação, o resultado é numérico (contínuo ou discreto). Por exemplo, um problema de classificação seria "*Conceder ou não crédito para um cliente?*" enquanto um problema de regressão seria "*Conceder qual valor de crédito para um cliente?*". As tarefas como preparação da base de dados, separação em conjuntos de treino e teste, definição dos critérios de parada do algoritmo, treinamento e teste são feitas de forma equivalente para ambos os problemas.

Assim como na classificação, a regressão consiste em realizar aprendizado supervisionado a partir de dados históricos. Além do tipo do resultado, a principal diferença entre os dois problemas está na avaliação de saída: na regressão, em vez de se estimar a acurácia, estima-se a distância ou o erro entre a saída do estimador (modelo) e a saída desejada. A saída de um estimador é um valor numérico contínuo que deve ser o mais próximo possível do valor desejado, e a diferença entre esses valores fornece uma medida de erro de estimação do algoritmo.

Para entender em linhas gerais o que é um problema de regressão, considere um grupo varejista com esta região de negócios:

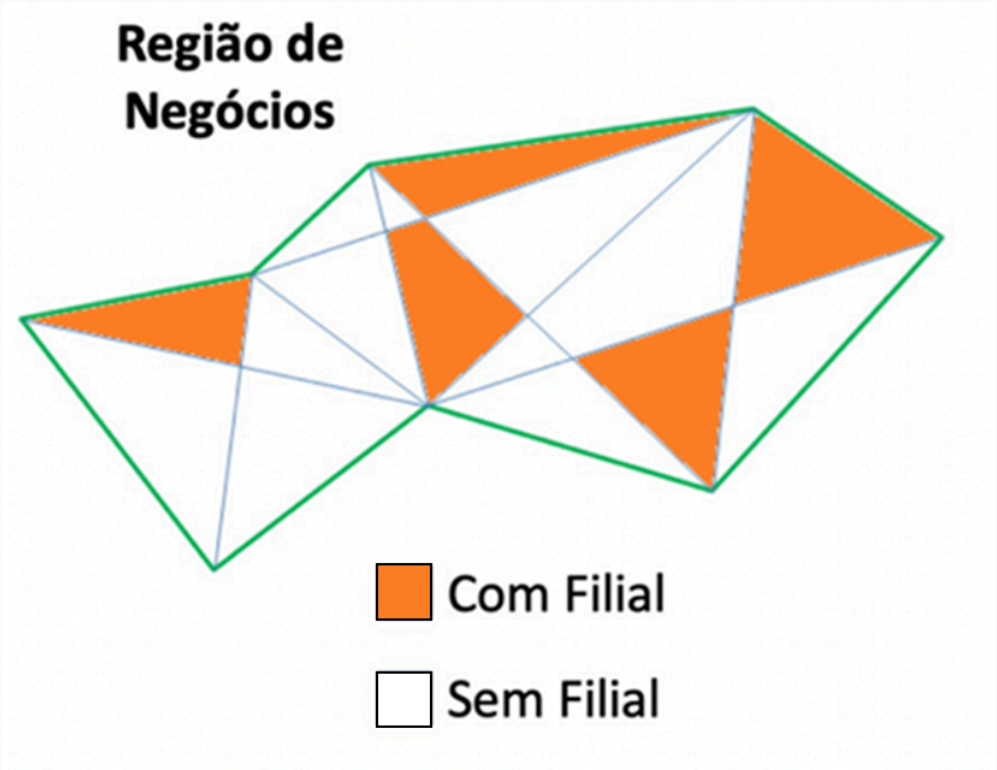


Figura 10.9: Exemplo de problema de regressão.

Esse grupo deseja expandir suas filiais, mas tem o seguinte problema: onde abrir uma nova filial? A resposta para essa pergunta pode ser determinada usando como métrica seu Faturamento Médio Anual. Essa métrica é conhecida nas regiões onde o grupo já tem filiais, mas desconhecida onde não tem:

Bairro	Faturamento
A	105.000
B	NA
C	NA
D	NA
...	...
K	NA
L	150.000
M	NA

Figura 10.10: Bairros e faturamentos.

Um problema de regressão seria estimar estes valores de regiões em potencial para novas filiais. Para tal, o primeiro passo seria levantar variáveis que estão presentes tanto nos bairros com e sem filiais, tais como: renda per capita, IDH, número de concorrentes, número de habitantes, preço do m^2 etc. A seguir, separamos nossos dados em dois conjuntos, com e sem faturamento, como ilustra a tabela a seguir:

	Bairro	Renda/Hab.	...	Preço do m^2	Faturamento
Com	A	1500	...	2500	105.000
	H	2400	...	5300	180.000
	...	...	...	...	...
	L	3400	...	2750	150.000
Sem	B	2500	...	1780	NA
	C	1000	...	3500	NA
	D	4300	...	6500	NA
	...	...	...	...	...
	K	7000	...	8900	NA
	M	2800	...	3900	NA

Figura 10.11: Bairros, características e faturamentos.

Com esses dados em mãos, basta elaborarmos um modelo de regressão para obter os valores estimados para as regiões sem filiais:

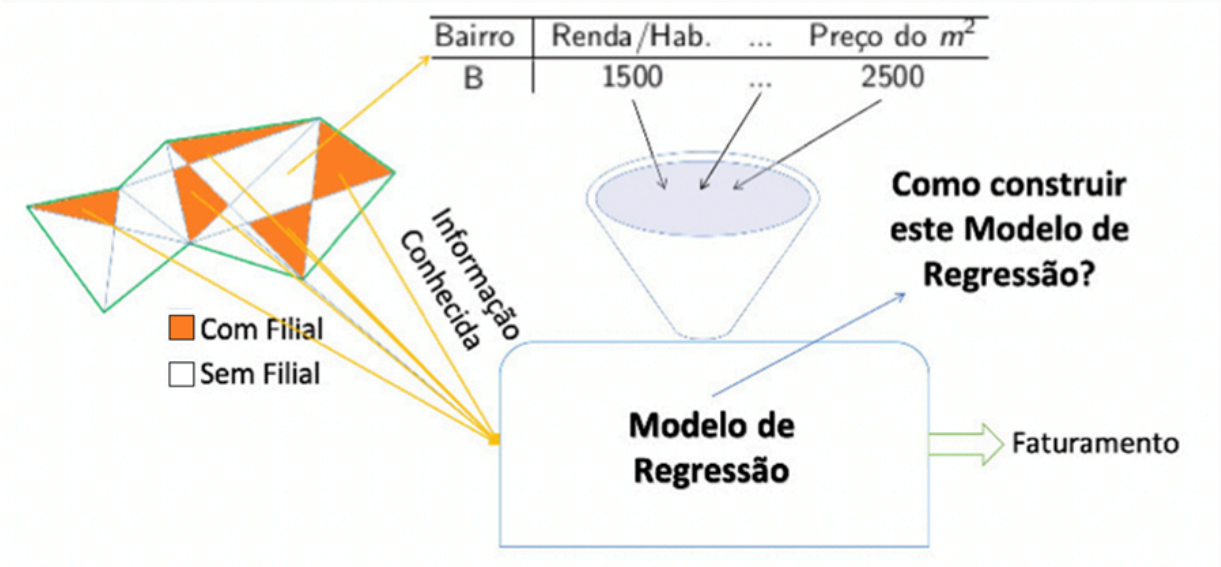


Figura 10.12: Modelo de regressão.

Finalmente, a partir do modelo, seleciona-se o bairro com Maior Faturamento Esperado para abrir uma nova filial:

	Bairro	Renda/Hab.	...	Preço do m^2	Faturamento
Observado	A	1500	...	2500	105.000
	H	2400	...	5300	180.000
	...	...	...	...	...
	L	3400	...	2750	150.000
Esperado	B	2500	...	1780	125.000
	C	**1000**	...	**3500**	**200.000**
	D	4300	...	6500	
	...	...	...	...	...
	K	7000	...	8900	180.000
	M	2800	...	3900	120.000

Figura 10.13: Bairros e faturamentos observados e esperados.

Assim, podemos definir um problema de Regressão como: dado um conjunto de *n* padrões, onde cada um é composto por informação de variáveis explicativas (independentes) e de uma variável resposta contínua ou discreta (dependente), tem-se como objetivo construir um modelo de regressão que, dado um novo padrão, estime o valor mais esperado para a variável resposta. A tabela a seguir ilustra o problema de regressão:

Padrão	Explicativas			Resposta
$\mathbf{x}_1$	x_{11}	...	x_{1J}	y_1
$\mathbf{x}_2$	x_{21}	...	x_{2J}	y_2
$\mathbf{x}_3$	x_{31}	...	x_{3J}	y_3
...	...	...	...	...
$\mathbf{x}_i$	x_{i1}	...	x_{iJ}	y_i
...	...	...	...	...
$\mathbf{x}_n$	x_{n1}	...	x_{nJ}	y_n

Figura 10.14: Problema de regressão.

Formalmente, seja d_j a resposta desejada para o objeto j e y_j a resposta predita do algoritmo, obtida a partir da entrada x_j; então, $e_j = d_j - y_j$ é o erro observado na saída do sistema para o objeto j. O processo de treinamento do estimador tem por objetivo corrigir este erro observado e, para isso, busca minimizar um critério (função objetivo) baseado em e_j, de maneira que os valores de y_j estejam próximos dos de d_j no sentido estatístico.

Métricas de avaliação

Existem diversas métricas de avaliação para problemas de regressão, todas elas buscando quantificar o erro do modelo, que representa a diferença entre o valor real e a saída do modelo, dado por:

$$e = y_j - \hat{y}_j$$

Figura 10.15: Erro.

Uma das métricas de avaliação mais usadas é a **MSE** (*Mean Squared Error*, ou *erro quadrático médio*). Quanto mais próximo de zero, melhor é o modelo de regressão analisado. Essa métrica fornece uma ideia da magnitude do erro, mas não da direção do mesmo, e sua fórmula é dada por:

$$MSE = \frac{1}{n}\sum_{j=1}^{n} e_j^2$$

Figura 10.16: MSE.

Baseada na MSE, temos a **RMSE** (*Root Mean Squared Error*, ou *raiz do erro quadrático médio*), que também quanto menor, melhor é o modelo. A raiz quadrada faz com que o erro seja medido na mesma unidade da saída do problema (variável *target*), e sua fórmula é dada por:

$$RMSE = \sqrt{\frac{1}{n}\sum_{j=1}^{n} e_j^2}$$

Figura 10.17: RMSE.

Outra métrica muito utilizada é o **coeficiente de determinação**, ou R_2 (*R-square*), que explica o quanto a variável *target* pode ser explicada pelo modelo (ou seja, o quanto *y* é explicado por *X*). Varia entre 0 e 1, e quanto mais próxima de 1, melhor o ajuste do modelo aos dados. Sua fórmula é dada por:

$$R^2 = 1 - \frac{SQE}{\sum_{j=1}^{n}(y_j - \bar{y})^2}$$

Figura 10.18: Coeficiente de determinação.

Sendo **SQE** a *Soma dos Quadrados dos Erros* (*Sum of Squared Errors*), calculada por:

$$SQE = \sum_{j=1}^{n} e_j^2$$

Figura 10.19: SQE.

10.3 Algoritmos de Machine Learning

A seguir, serão apresentados alguns algoritmos de *Machine Learning* que podem ser usados para problemas de classificação e regressão.

KNN (classificação e regressão)

O algoritmo KNN (*k-Nearest Neighbors* ou, em português, k-Vizinhos Mais Próximos) é simples de entender e funciona muito bem na prática tanto para problemas de classificação quanto para problemas de regressão. Este é um algoritmo não paramétrico, ele não assume premissas sobre a distribuição dos dados. Sua ideia principal é considerar que os exemplos vizinhos são similares ao exemplo cuja informação se deseja inferir.

O KNN utiliza uma métrica de distância para encontrar as *k* instâncias mais semelhantes nos dados de treinamento para uma nova instância e considera o resultado dos vizinhos como a previsão (a classe mais comum, em problemas de classificação, ou a média da variável *target*, em problemas de regressão).

A figura a seguir ilustra essa ideia para problemas de classificação: o KNN considera que os registros do conjunto de dados correspondem a pontos no R_n, em que cada atributo corresponde a uma dimensão deste espaço.

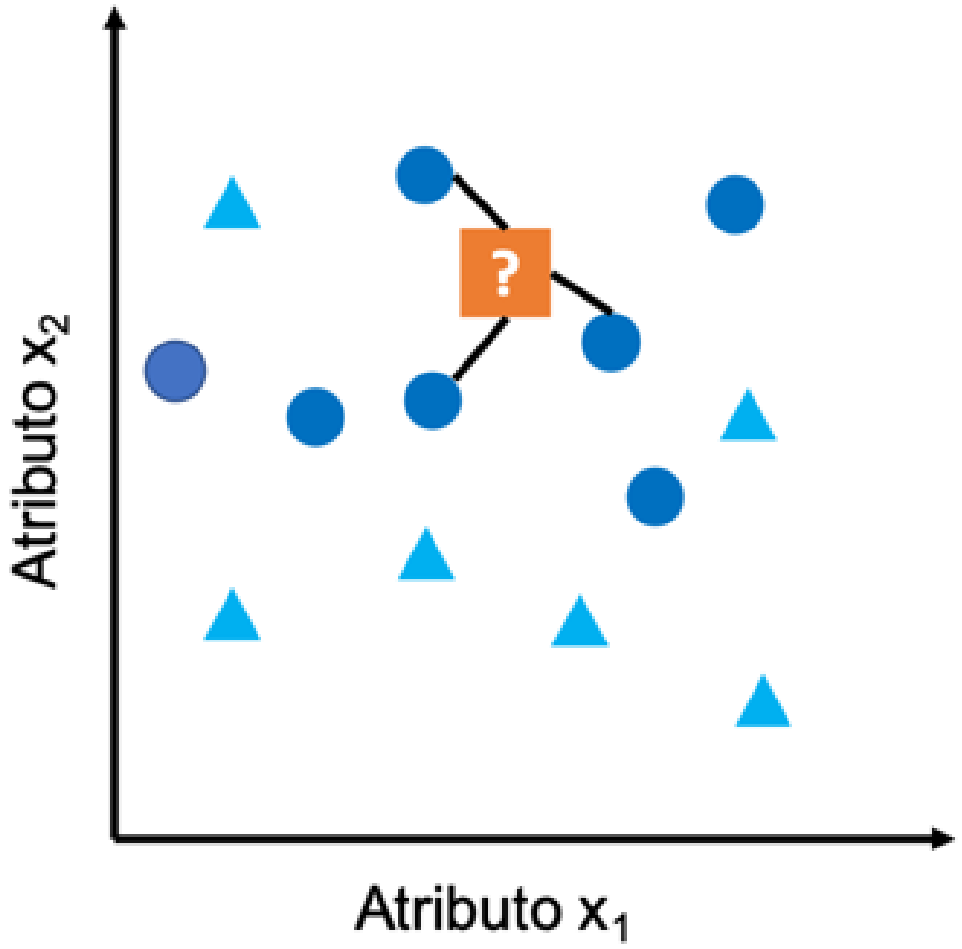


Figura 10.20: Funcionamento do KNN.

No KNN, inicialmente, o conjunto de dados (rotulado) é armazenado. Quando um novo registro deve ser classificado, ele é comparado a todos os registros do conjunto de treinamento para identificar os *k* (parâmetro de entrada) vizinhos mais próximos (mais semelhantes) de acordo com alguma métrica de distância (por exemplo, distância euclidiana). A classe do novo registro é determinada por inspeção das classes desses vizinhos mais próximos, de acordo com a métrica escolhida. Na maioria das implementações do KNN, os atributos são normalizados, para que tenham a mesma contribuição na predição da classe ou do valor.

As etapas a seguir resumem o algoritmo KNN:

1. Definição da métrica de distância e valor de *k*;
2. Cálculo da distância do novo registro a cada um dos registros existentes no conjunto de referência;
3. Identificação dos *k* registros do conjunto de referência que apresentaram menor distância em relação ao novo registro (mais similares);
4. Apuração da classe mais frequente entre os *k* registros identificados no passo anterior (usando votação majoritária), em problemas de classificação, ou da média aritmética dos *k*-vizinhos mais próximos, em problemas de regressão.

Apesar de ser um algoritmo muito utilizado, o KNN tem algumas limitações: a performance de predição pode ser lenta em *datasets* grandes; é sensível a características irrelevantes, uma vez que todas as características contribuem para o cálculo da distância e consequentemente, para a predição; e é necessário testar diferentes valores de *k* e a métrica de distância a utilizar.

Árvore de decisão (classificação e regressão)

A **árvore de decisão** é um dos modelos preditivos mais simples de ser interpretado, e é inspirado na forma como os humanos tomam decisões. Uma de suas principais vantagens é apresentar a informação visualmente de uma forma fácil de entender pelo ser humano. As árvores podem ser usadas para problemas de classificação (nesse caso, chamadas de ***árvores de classificação***) ou de regressão (chamadas ***árvores de regressão***). Basicamente, uma árvore de decisão usa amostras das características dos dados para criar regras de decisão no formato de árvore, isto é, reduz os dados em um conjunto de regras que podem ser usadas para uma decisão.

As árvores de decisão aliam acurácia e interpretabilidade. Elas possibilitam a seleção automática de variáveis para compor suas estruturas: cada nó interno representa uma decisão sobre um atributo que determina como os dados estão particionados pelos seus nós filhos. Para fazer a predição de um novo exemplo, basta

testar os valores dos atributos na árvore e percorrê-la até se atingir um nó folha (resultado predito). A figura a seguir ilustra uma árvore de classificação.

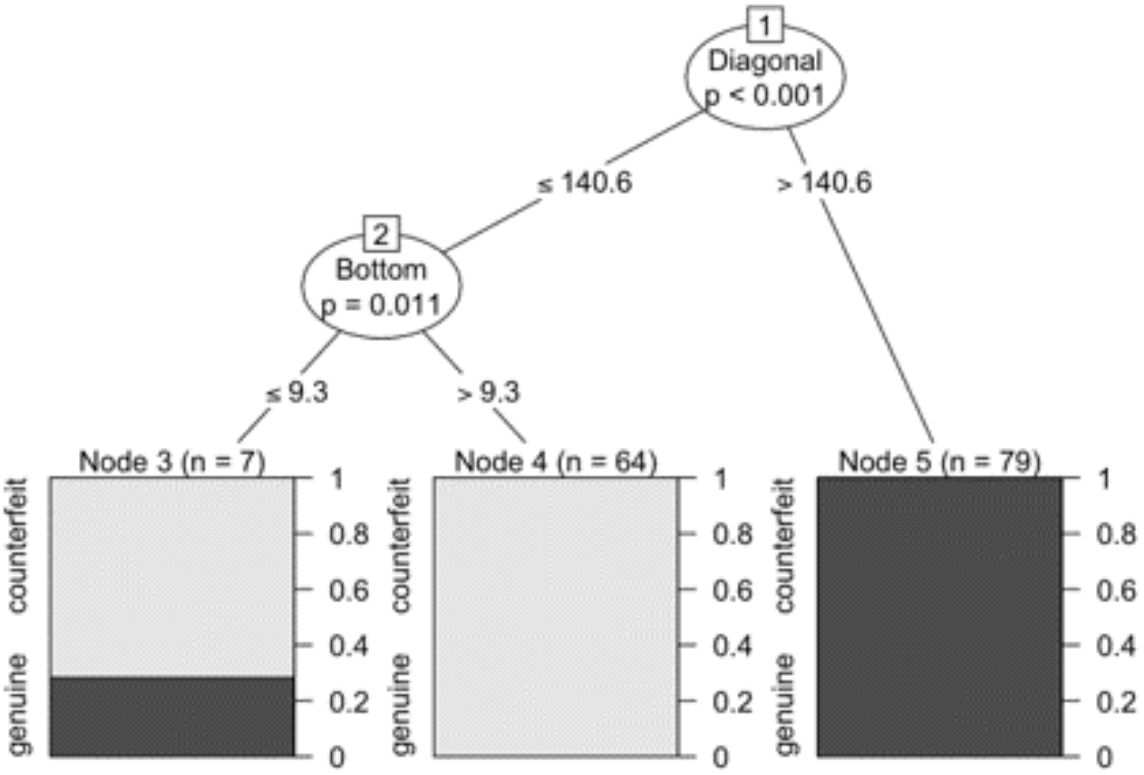


Figura 10.21: Funcionamento da árvore de classificação.

As árvores de regressão são muito similares às árvores de classificação, com a diferença de que, nas árvores de regressão, a predição (ou valor estimado) é a média dos valores dos exemplos de cada folha, enquanto nas árvores de classificação a predição é a classe mais frequente nos exemplos de cada folha. Ambas são construídas de forma similar: a partir do nó raiz, os dados são particionados usando uma estratégia de divisão e conquista de acordo com a característica que apresentará o resultado mais homogêneo após a separação ser realizada. Enquanto nas árvores de classificação a homogeneidade é medida pela entropia (ou medidas similares), nas árvores de regressão, a homogeneidade é

medida por estatísticas como variância, desvio padrão ou desvio absoluto da média.

Um critério de divisão comum para árvores de regressão é a redução de desvio padrão (SDR – *Standard Deviation Reduction*). Essa fórmula mede a redução no desvio padrão, comparando o desvio padrão antes da divisão com o desvio padrão ponderado após a divisão. Por exemplo, se o desvio padrão é mais reduzido com a divisão da árvore *T* em uma característica *B* do que em *A*, deve-se realizar a divisão primeiro em *B*, resultando em uma árvore mais homogênea. Supondo que a árvore de regressão está pronta com apenas esta divisão em *B*, a predição pode ser feita considerando se o novo exemplo caiu no conjunto *T1* ou no conjunto *T2* e calculando a média dos valores desse conjunto.

Há diferentes algoritmos para a elaboração de uma árvore de decisão. Alguns exemplos são: *ID3*, *CTree*, *C4.5*, *C5.0* e *CART*. Todos os algoritmos são bem parecidos: em geral, a construção da árvore é realizada de acordo com alguma abordagem recursiva de particionamento do conjunto de dados; a principal distinção está nos processos de seleção de variáveis, no critério de particionamento e no critério de parada para o crescimento da árvore.

Embora sejam intuitivas e interpretáveis, as árvores de decisão por si só não costumam ter o mesmo nível de precisão preditiva que alguns outros algoritmos populares de aprendizado de máquina. Elas tendem a criar limites de decisão excessivamente complicados, resultando em maior variação do modelo, o que leva ao *overfitting*. Para evitar esse problema na prática, podemos usar algoritmos de poda (*pruning*) para reduzir a profundidade e a complexidade da árvore, removendo nós maiores que uma certa profundidade, para que a árvore de decisão não se ajuste em excesso aos dados de treinamento.

Árvore C4.5

Entre os algoritmos de árvores de classificação existentes, um dos mais conhecidos é o C4.5, que utiliza conceitos e medidas de Teoria da Informação, uma disciplina que estuda a quantificação, armazenamento e comunicação da informação.

O funcionamento do C4.5 pode ser resumido assim: inicialmente, a raiz da árvore contém todo o conjunto de dados com exemplos misturados de várias classes. Um *predicado* (ponto de separação) é escolhido como sendo a condição que melhor separa ou discrimina as classes. O predicado é um dos atributos previsores do problema e induz uma divisão do conjunto de dados em dois ou mais conjuntos disjuntos, cada um deles associado a um nó filho. Cada novo nó abrange um subconjunto do conjunto de dados original que é recursivamente separado até que o subconjunto associado a cada nó folha consista inteira ou predominantemente de registros de uma mesma classe.

As etapas a seguir resumem o algoritmo C4.5 para problemas de classificação:

- Calcular a entropia do conjunto *T* completo;
- Para cada atributo:
    - Calcular o ganho de informação;
    - Selecionar o atributo com maior ganho de informação para o nó raiz da árvore;
    - Subdividir o conjunto *T*.
- Repetir o procedimento para cada nó gerado.

***Naïve Bayes* (classificação)**

O ***Naïve Bayes***, ou *Bayes Ingênuo*, é um classificador genérico e de aprendizado dinâmico. É um dos métodos mais utilizados para classificação – especialmente em aplicações de *text mining*, previsões em tempo real e/ou em sistemas embarcados –, pois é rápido computacionalmente e só necessita de um pequeno número de dados de treinamento. Ele é especialmente adequado quando o problema tem um grande número de atributos (características), e

determina a probabilidade de um exemplo pertencer a uma determinada classe.

Esse método é chamado de *ingênuo* (*naïve*, em inglês) porque desconsidera completamente a correlação entre os atributos (características), tratando cada um de forma independente. Além disso, o nome do método contém a palavra Bayes porque é baseado no Teorema de Bayes, que determina a probabilidade de um evento com base em um conhecimento prévio (a priori) que pode estar relacionado a este evento.

Seja *X* uma observação (ou evidência) proveniente de um conjunto de *n* atributos; *H*, a hipótese de que a observação *X* pertença a uma determinada classe *C*. Em problemas de classificação, queremos determinar *P(H|X)*, ou seja, a probabilidade de a hipótese *H* se aplicar a *X*. Em outras palavras, buscamos a probabilidade de que a observação *X* pertença à classe *C*, dado que conhecemos a descrição de *X*. Assim:

- *P(H|X)* é denominada a probabilidade a posteriori de *H* condicionada a *X*. Por exemplo, suponha que tenhamos os dados de idade e renda mensal de clientes. *X* é um cliente de 30 anos com renda mensal R$ 5.000,00, e *H* é a hipótese de este cliente comprar um celular. *P(H|X)* é então a probabilidade de este cliente *X* comprar um celular, dado que conhecemos sua idade e renda mensal;
- *P(H)*, por sua vez, é a probabilidade a priori de *H*. É a probabilidade de qualquer cliente, independente de idade, renda mensal ou qualquer outra informação, comprar um celular. Essa probabilidade é baseada em mais informação do que *P(H)*, que é independente de *X*;
- De forma análoga, *P(X|H)* é a probabilidade a posteriori de *X* condicionada a *H*. Por exemplo, a probabilidade de um cliente *X* ter 30 anos e ganhar R$ 5.000,00, dado que este cliente comprou um celular;
- *P(X)*, finalmente, é a probabilidade a priori de *X*, a probabilidade de que um cliente do nosso conjunto de dados tenha 30 anos e

ganhe R$ 5.000,00.

P(H), *P(X|H)* e *P(X)* podem ser estimadas a partir do próprio conjunto de dados. O Teorema de Bayes é útil para calcular a probabilidade a posteriori *P(H|X)* a partir de *P(H)*, *P(X|H)* e *P(X)*:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

Figura 10.22: Teorema de Bayes.

Finalmente, apresentamos a definição formal do algoritmo *Naïve Bayes*. Seja $X(A_1, A_2, \ldots, A_n, C)$ um conjunto de dados. Considere que $c_1, c_2, \ldots, c_k$ são as classes do problema (valores possíveis do atributo alvo *C*) e que *R* é um registro que deve ser classificado. Sejam ainda $a_1, a_2, \ldots, a_n$ os valores que *R* assume para os atributos previsores $A_1, A_2, \ldots, A_n$, respectivamente.

O algoritmo consiste em dois passos:

- Calcular as probabilidades condicionais $P(C=c_i|R)$, $i = 1, 2, \ldots, k$;
- Indicar como saída do algoritmo a classe *c* tal que *P(C=c|R)* seja máxima, quando considerados todos os valores possíveis do atributo-alvo *C*.

A intuição por trás do algoritmo é dar mais peso para as classes mais frequentes e, conforme já discutimos, é dito ingênuo porque considera como hipótese que os atributos são estatisticamente independentes entre si, o que em muitos casos práticos não ocorre. Entretanto, na prática, o método mostra-se bastante efetivo, mesmo nos casos em que os previsores não são estatisticamente independentes.

SVM (classificação e regressão)

O SVM (*Support Vector Machine*, ou máquina de vetor de suporte) é um dos algoritmos mais efetivos para classificação e que também pode ser utilizado para problemas de regressão, apesar de menos comum. O SVM pode ser aplicado em dados lineares ou não lineares. Embora o treinamento dos modelos de SVM costume ser lento, esses modelos exigem poucos ajustes, tendem a apresentar boa acurácia e conseguem modelar fronteiras de decisão complexas e não lineares. Além disso, são menos propensos a *overfitting* se comparados com outros métodos.

Essencialmente, o SVM realiza um mapeamento não linear para transformar os dados de treino originais em uma dimensão maior. Nessa nova dimensão, o algoritmo busca pelo hiperplano que separa os dados linearmente de forma ótima. Com um mapeamento apropriado para uma dimensão suficientemente alta, dados de duas classes podem ser sempre separados por um hiperplano. O SVM encontra esse hiperplano usando vetores de suporte (exemplos essenciais para o treinamento) e margens, definidas pelos vetores de suporte.

Para melhor entendimento do SVM aplicado a problemas de classificação (uso mais comum), considere um conjunto de dados de treinamento linearmente separável na forma $\{x_i, y_i\}$, em que x_i corresponde ao vetor de 2 atributos previsores e $y_i \in \{-1,1\}$, as duas classes possíveis do problema. O conjunto de dados de entrada é utilizado para construir uma função de decisão $f(x)$ tal que:

- Se $f(x) > 0$, então $y_i = 1$;
- Se $f(x) < 0$, então $y_i = -1$

A figura a seguir ilustra este exemplo.

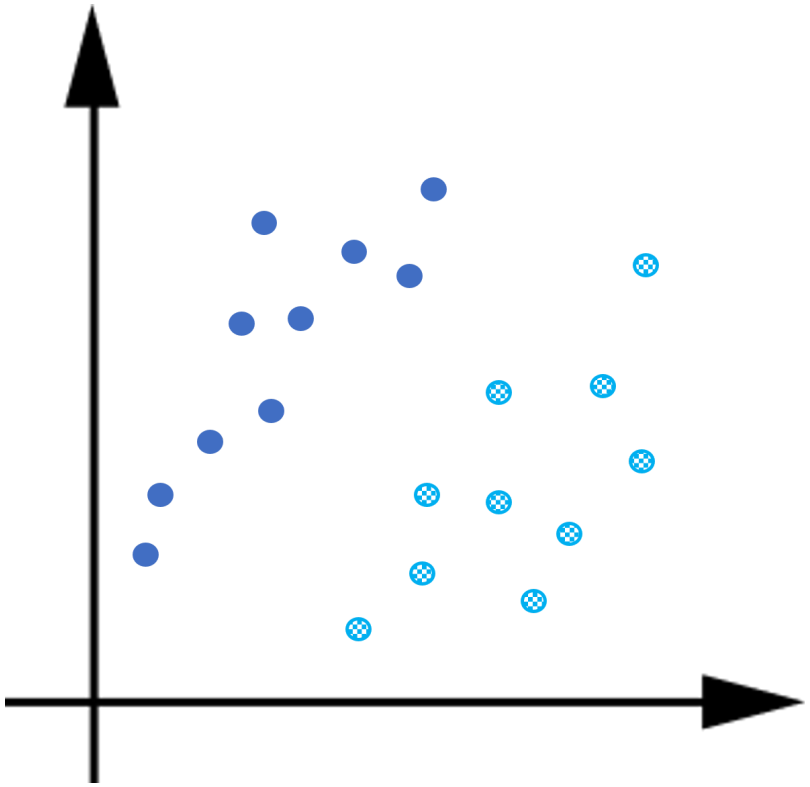


Figura 10.23: Dados de entrada.

Conforme já mencionado, o algoritmo SVM constrói classificadores lineares, que separam os conjuntos de dados por meio de um hiperplano. Considere hiperplano como a generalização do conceito de plano para dimensões maiores que 3. Em um espaço *p*-dimensional, um hiperplano é um subespaço achatado de dimensão *p-1* que não precisa passar pela origem. Quando *d = 2*, o hiperplano é uma reta e sua equação é dada por:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$$

Figura 10.24: Equação do plano.

Nessa equação, β_1 e β_2 são parâmetros da reta que determinam a sua inclinação e β_0, o parâmetro que determina o ponto de corte do eixo y. Neste caso, o propósito do SVM é determinar parâmetros da

reta, β_0, β_1 e β_2, que permitem separar os conjuntos dos dados de treinamento em duas classes possíveis:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 < 0$$

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 > 0$$

Figura 10.25: Duas classes possíveis.

Esse exemplo é ilustrado pela figura a seguir.

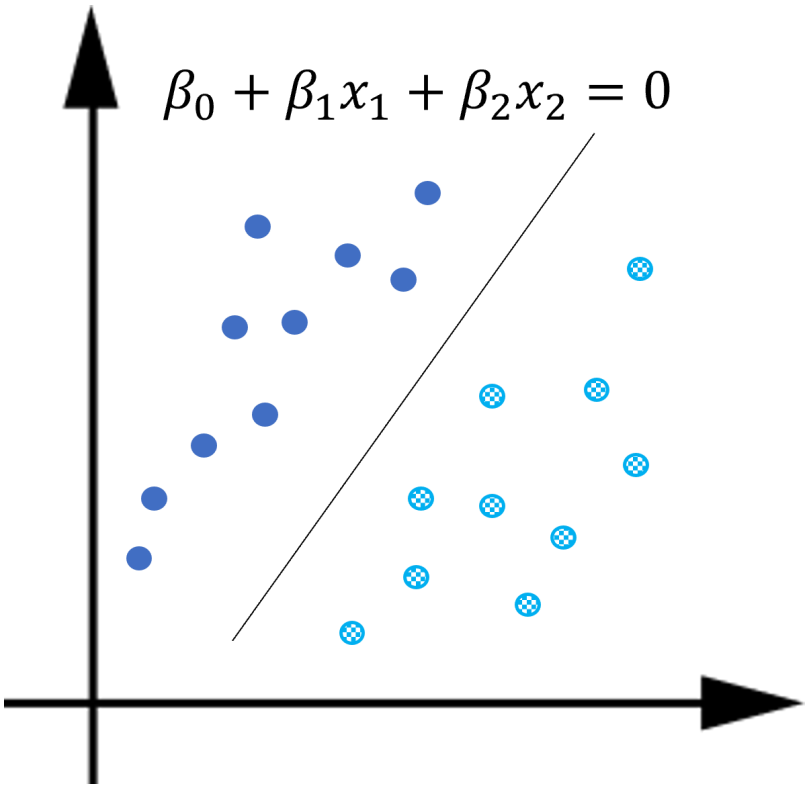


Figura 10.26: Classificação no SVM.

Da mesma forma que um plano tem 2 dimensões e divide um conjunto tridimensional em 2 espaços de dimensão 3, um hiperplano

em um espaço n-dimensional tem *n-1* dimensões e divide este espaço em 2 subespaços de dimensão *n*. Assim, para *d = p*, a equação do hiperplano é:

$$\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n = 0$$

Figura 10.27: Equação do hiperplano.

Neste caso, o hiperplano também divide o espaço *p*-dimensional em duas metades:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 < 0$$

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 > 0$$

Figura 10.28: Divisões do hiperplano.

Voltando ao exemplo anterior, como afirmamos previamente que o problema é linearmente separável, é possível construir um hiperplano que separe as observações de treino perfeitamente de acordo com seus rótulos de classe. Esse hiperplano tem as seguintes propriedades:

$$\beta_0 + \beta_1 x_i + \beta_2 x_2 > 0, se\ y_i = 1$$

$$\beta_0 + \beta_1 x_i + \beta_2 x_2 < 0, se\ y_i = -1$$

Figura 10.29: Propriedades do hiperplano.

O hiperplano pode então ser usado para construir um *classificador*: o exemplo de teste recebe a classe dependendo de que lado do hiperplano estiver localizado. É importante notar que, no exemplo anterior, infinitas retas dividem corretamente o conjunto de treinamento em duas classes, conforme ilustra a figura a seguir. O SVM deve, então, realizar um processo de escolha da reta separadora, dentre o conjunto infinito de retas possíveis:

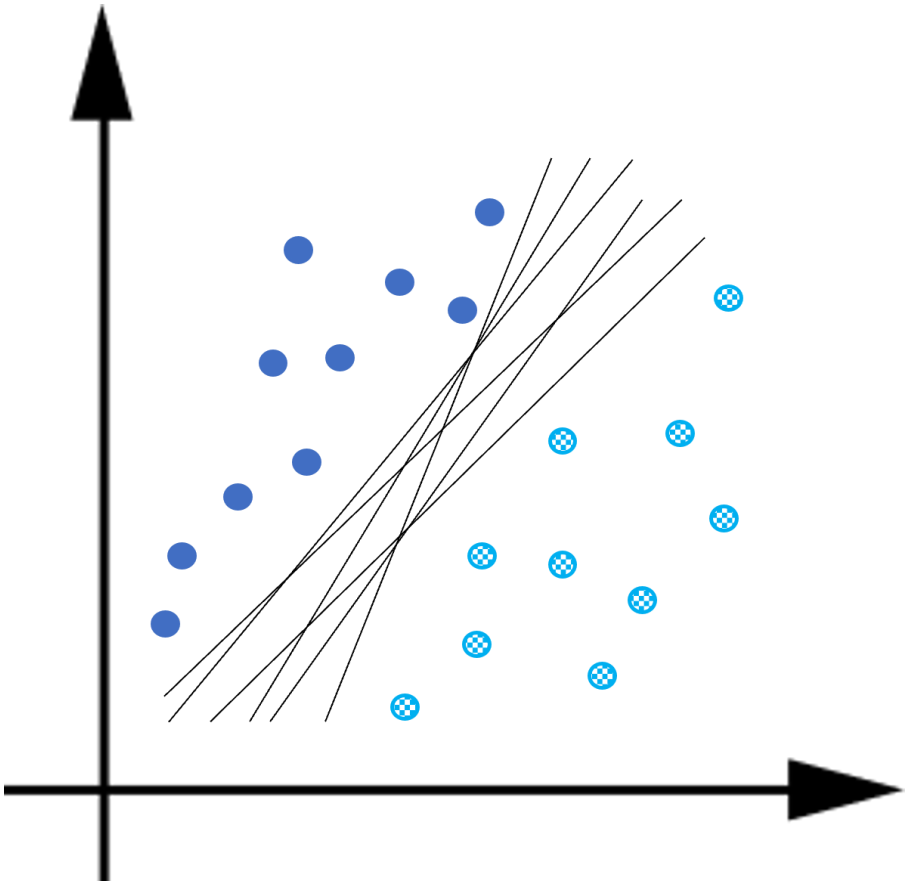


Figura 10.30: Múltiplas retas separadoras.

Na figura a seguir, é apresentado apenas um classificador linear (ilustrado pela reta sólida) e duas retas paralelas a este classificador, pontilhadas. Cada uma das retas pontilhadas é movida a partir da posição da reta sólida e determina quando a reta paralela intercepta o primeiro ponto do conjunto de dados. A *margem* é a distância construída entre essas duas retas paralelas pontilhadas:

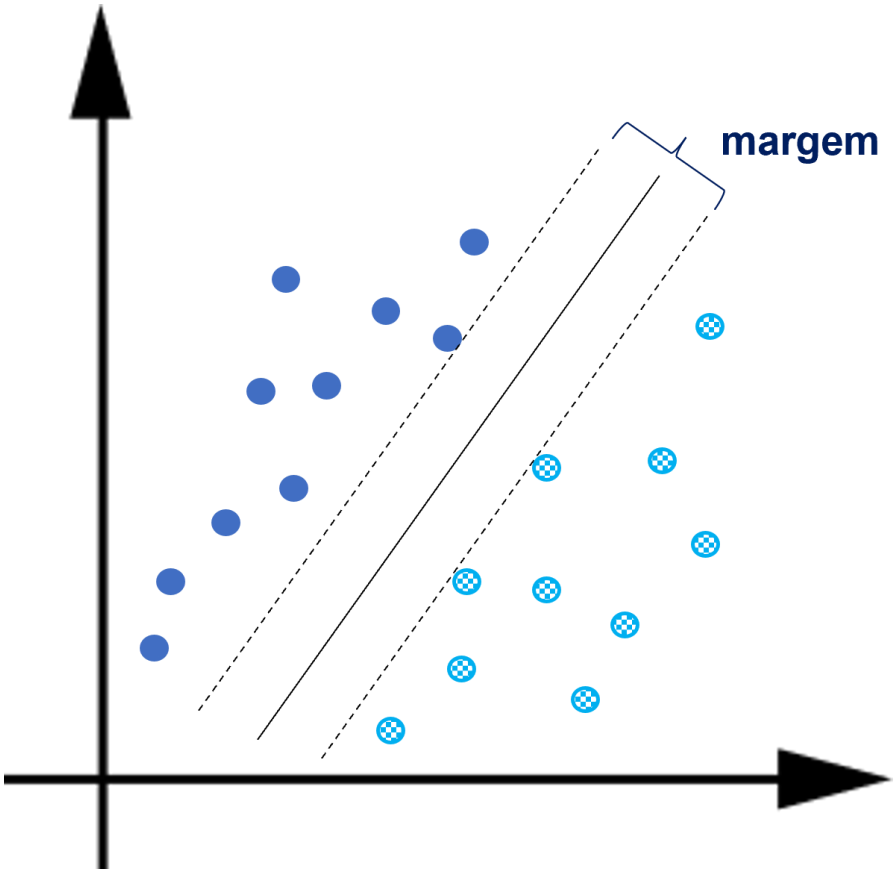


Figura 10.31: Conceito de margem no SVM.

Assim como existem infinitas retas que separam os pontos em duas classes, há diversos tamanhos de margem possíveis dependendo da reta escolhida como classificador. A figura a seguir ilustra dois possíveis classificadores, com dois tamanhos de margem diferentes.

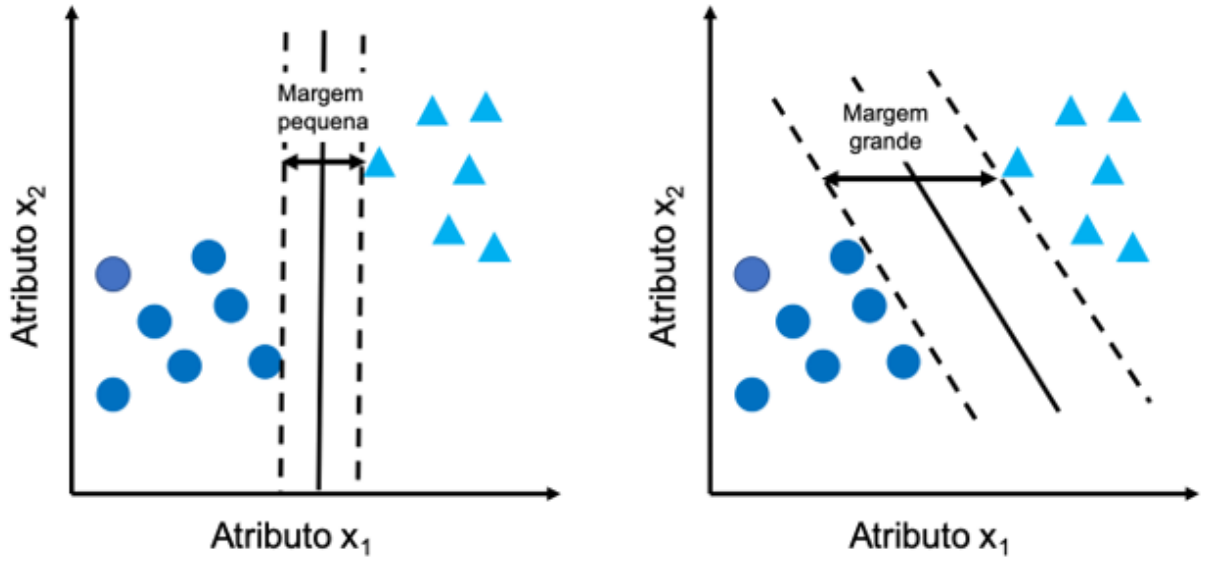


Figura 10.32: Diferentes tamanhos de margem no SVM.

O classificador associado ao valor máximo de margem é denominado *classificador linear de margem máxima*. Os pontos do conjunto de dados de treinamento que são interceptados pelas linhas da margem são denominados *vetores de suporte* e são os pontos mais difíceis de classificar. Os vetores de suporte são ilustrados pela figura a seguir. Por construção, todos os vetores de suporte possuem a mesma distância em relação à reta do classificador linear (a metade do comprimento da margem). É importante ressaltar que, apesar de o classificador linear de margem máxima ser geralmente bom, pode haver *overfitting* se o número de dimensões for grande.

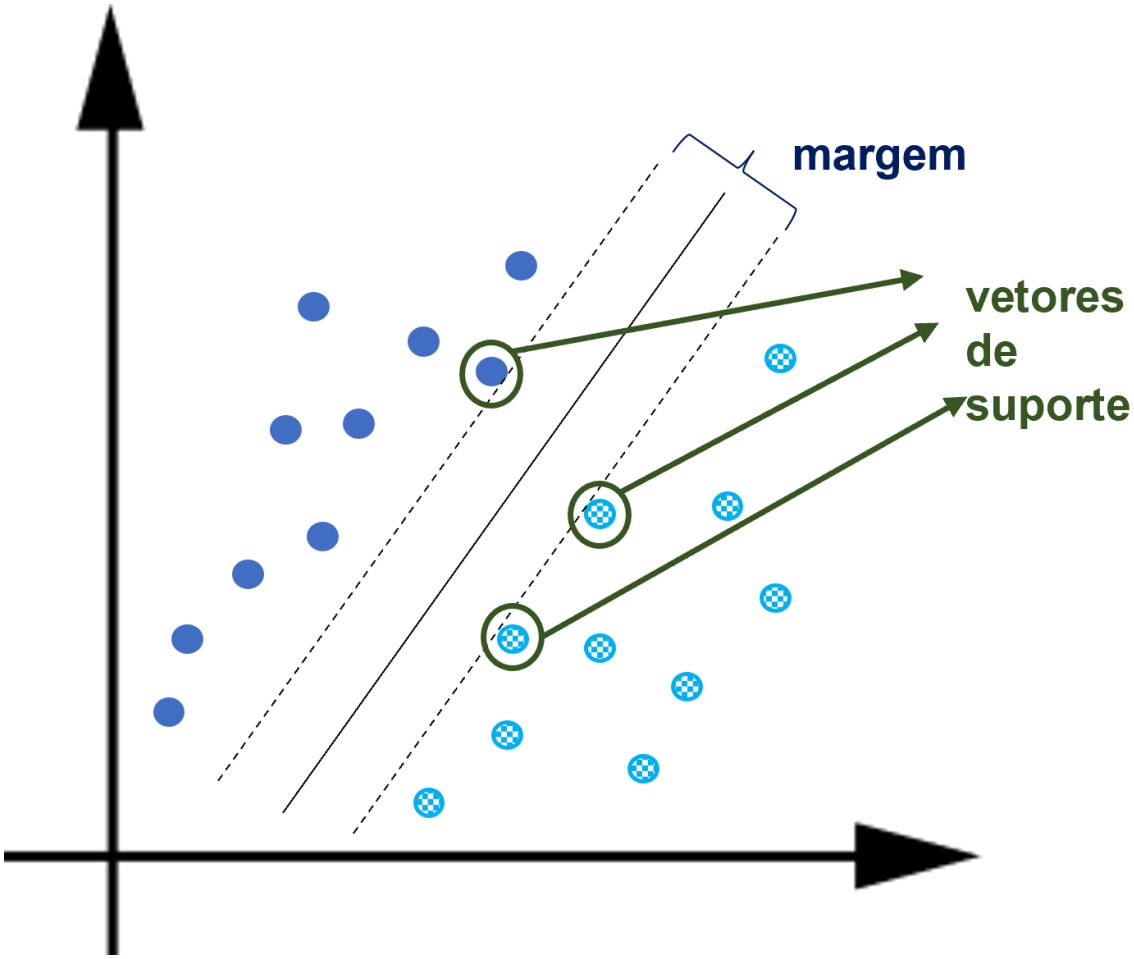


Figura 10.33: Conceito de vetores de suporte no SVM.

O SVM realiza um processo de otimização pelo qual são determinados os parâmetros do classificador linear (β_0, β_1 e β_2). O objetivo desse processo de otimização é determinar os valores de β_0, β_1 e β_2 que produzam o valor máximo para o comprimento da margem.

A reta correspondente ao classificador linear é dita *ótima* porque, se ela for deslocada em alguma das duas direções das retas perpendiculares a ela, a probabilidade de haver um erro de classificação é maior. Assim, a posição do classificador linear correspondente ao comprimento de margem máximo é a mais segura possível com relação a eventuais erros de classificação, e

quanto maior a distância de *x* para o hiperplano, maior a confiança sobre a classe a que *x* pertence.

A solução do problema de otimização do SVM pode ser obtida usando técnicas de programação quadrática, muito conhecidas na área de Pesquisa Operacional, que consistem em otimizar uma função quadrática sujeita a restrições lineares. A solução desse problema está fora do escopo deste livro.

Uma vez obtidos os valores dos parâmetros β_0, β_1 e β_2, aplica-se uma função de decisão para classificar um novo exemplo x_i. A classe de x_i, y_i, é dada pelo sinal de $f(x)$:

$$Se\ f(x_i) > 0, então\ y_i = 1$$

$$Se\ f(x_i) < 0, então\ y_i = -1$$

Figura 10.34: Função de classificação.

Na prática, entretanto, os dados reais não costumam ser perfeitamente separáveis por um hiperplano. Além disso, o **hiperplano margem máxima** é extremamente sensível a mudanças em uma única observação, o que sugere que pode ocorrer *overfitting* nos dados de treino. Por isso, podemos querer considerar um classificador baseado em um hiperplano que não separe perfeitamente as duas classes, com o objetivo de aumentar a robustez nas observações individuais e melhorar a classificação na maioria das observações de treino. Ou seja, pode valer a pena classificar erroneamente algumas observações de treino a fim de melhorar a classificação nas observações restantes.

Para isso, usamos o classificador *soft-margin*, que permite que algumas observações do conjunto de treino violem a linha de separação. Neste caso, o hiperplano é escolhido para separar

corretamente a maior parte das observações em duas classes, mas pode classificar incorretamente algumas observações. Um conjunto adicional de coeficientes é introduzido na otimização para permitir uma "folga" à margem; em contrapartida, aumentam a complexidade do modelo.

No exemplo ilustrado pela figura a seguir, o classificador cometeria erros na classificação para os dois pontos destacados, que podem ser considerados ruído. Para isso, o parâmetro de custo *C* define a "rigidez" da margem e controla o *trade-off* entre o tamanho da margem e o erro do classificador. Quanto maior o valor de *C*, mais pontos podem ficar dentro da margem e maior o erro de classificação, mas menor é a chance de *overfitting*. Se *C = 0*, a margem é rígida e temos o classificador de margem máxima. Na prática, o classificador com margem excessivamente rígida não produz bons resultados, pois dificulta a generalização do modelo.

Observação: é importante ressaltar que na biblioteca Scikit-learn o parâmetro *C* representa um parâmetro de regularização e funciona de forma contrária, adicionando uma penalidade a cada ponto classificado erradamente. Se o valor de *C* for pequeno, então, essencialmente, a penalidade para pontos classificados erradamente também será pequena, resultando em uma margem maior (mais flexível).

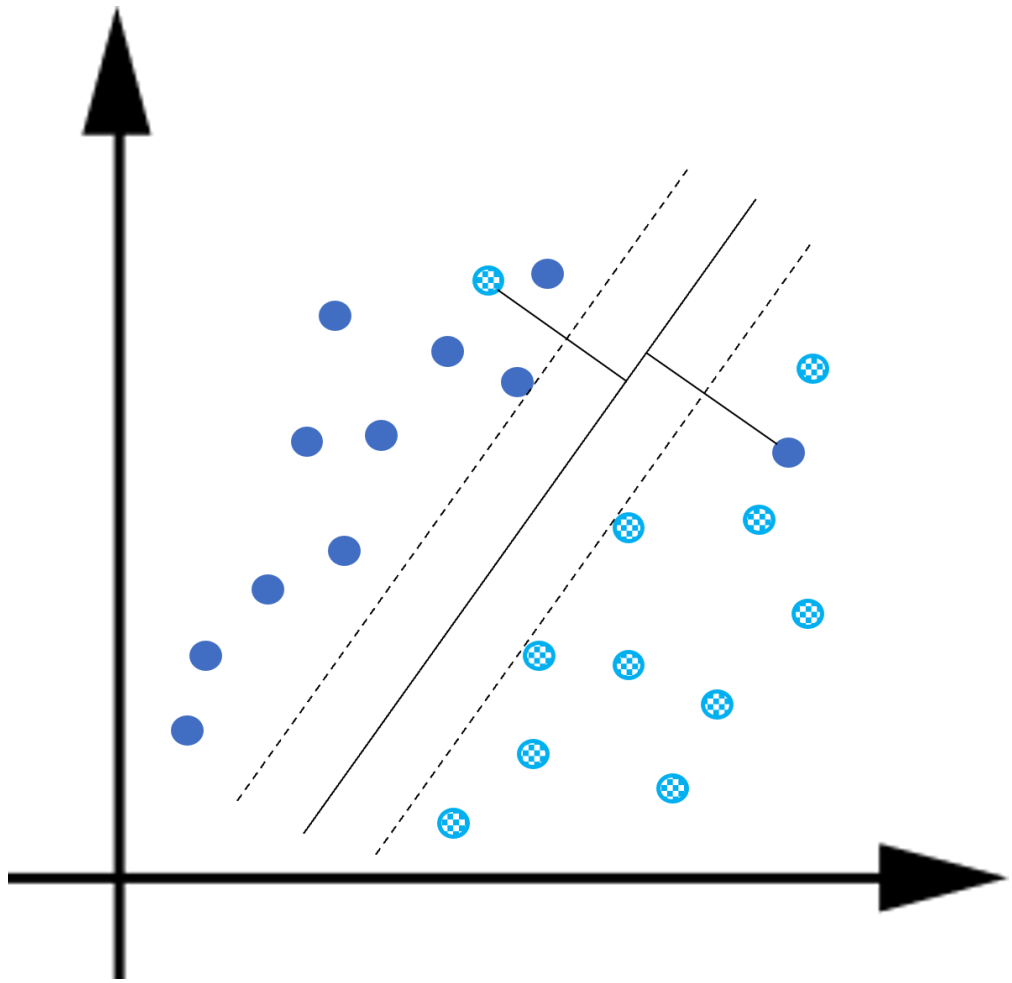


Figura 10.35: Classificador de margem máxima.

O número total de vetores de suporte depende da quantidade de folga permitida nas margens e da distribuição dos dados. Quanto maior a folga permitida, maior o número de vetores de suporte e mais lenta será a classificação dos dados de teste, pois a complexidade computacional do SVM está relacionada com o número de vetores de suporte.

Na prática, o SVM é implementado usando funções ***kernel***, que, de forma simplificada, são objetos matemáticos que permitem que trabalhemos em um espaço de dimensão maior. Neste caso, em vez de se utilizar as observações em si, é utilizado o seu produto interno. A previsão de um novo exemplo é feita calculando o produto escalar

entre o exemplo (x) e cada vetor de suporte (x_i). Os tipos de *kernel* mais utilizados são o *linear*, o *polinomial* e o *radial*.

Para um conjunto de dados que não é linearmente separável, o SVM utiliza funções *kernel* para mapear o conjunto de dados para um espaço de dimensão maior que a original. O classificador é então ajustado neste novo espaço. O SVM é, na verdade, a combinação do classificador linear com um *kernel* não linear. Esse processo é ilustrado pela figura a seguir.

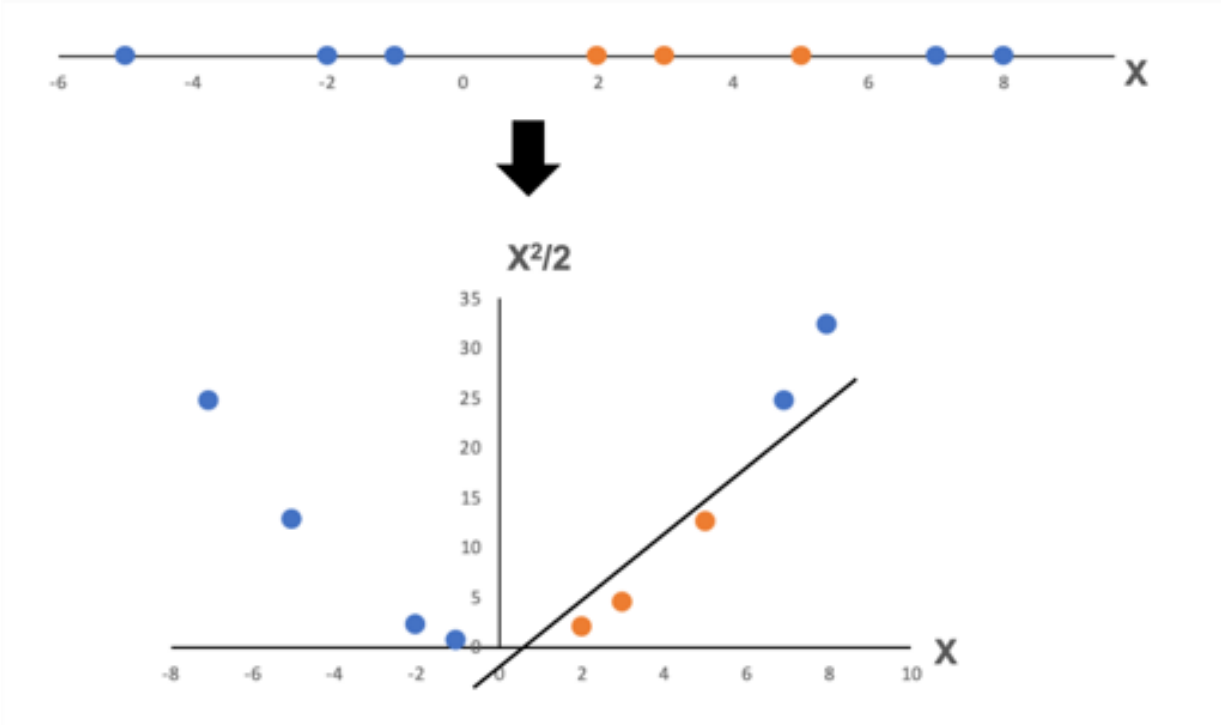


Figura 10.36: Mapeamento do conjunto de dados em um espaço de dimensão maior.

O SVM também pode ser aplicado a problemas de classificação que envolvem múltiplas classes. Neste caso, usa-se o algoritmo para treinar um modelo de classificação que informe se o registro é da classe c_i (região positiva) ou se é de alguma outra classe diferente de c_i (região negativa). Dessa forma, constrói-se p-1 modelos de classificação, onde p é o número de classes possíveis no problema. Para classificar um novo exemplo, pode-se submetê-lo a cada um

dos modelos de classificação gerados, e a classe selecionada para o exemplo é a mais frequentemente atribuída.

O SVM também pode ser usado como método de regressão, sendo chamado **SVR** (*Support Vector Regression*). O SVR foi proposto em 1997, mas é pouco utilizado, pois existem modelos mais simples para regressão com resultados semelhantes ou melhores. Assim como no SVM para classificação, o modelo produzido pelo SVR depende apenas de um subconjunto dos dados de treinamento.

A ideia básica do SVR é mapear um conjunto de dados X em um espaço multidimensional, através um mapeamento não linear (usando funções *kernel*), e realizar uma *regressão linear* neste espaço transformado, considerando apenas os pontos que estão dentro da margem. O melhor modelo é o hiperplano que possui o número máximo de pontos.

Comparado ao modelo de regressão linear, o SVR tem a vantagem de utilizar uma grande variedade de funções que se adequam aos modelos. Na regressão linear, tentamos minimizar a taxa de erro, enquanto no SVR, tentamos ajustar o erro dentro de um determinado limite, definido pela margem.

Regressão linear (regressão)

A **regressão linear**, como o nome já diz, é um algoritmo para o problema de regressão. Formalmente, esse algoritmo modela a relação entre a variável de resposta (y) e as variáveis preditoras (X). A regressão corresponde ao problema de estimar uma função a partir de pares entrada-saída e considera que y pode ser explicado por uma combinação linear de X. Quando temos apenas um x em X (uma única característica), temos uma *regressão linear simples*, cuja equação é $y = \beta_0 + \beta_1 x$, sendo β_0 e β_1 os coeficientes de regressão (especificam, respectivamente, o intercepto do eixo y e a inclinação da reta). Para o caso da *regressão linear múltipla*, a equação deve

ser estendida para equação de plano/hiperplano: $y = \beta_0 + \beta 1_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$.

A solução da tarefa de regressão consiste em encontrar valores para os coeficientes de regressão de forma que a reta (ou plano/hiperplano) se ajuste aos valores assumidos pelas variáveis no conjunto de dados. Além do *método dos mínimos quadrados*, existem diversas técnicas matemáticas para determinar os coeficientes, que estão fora do escopo deste livro.

A saída do estimador é um valor numérico contínuo que deve ser o mais próximo possível do valor desejado, e a diferença entre esses valores fornece uma medida de erro de estimação do algoritmo. Seja d_j a resposta desejada para o objeto j e y_j a resposta predita do algoritmo, obtida a partir da entrada x_j, então $e_j = d_j - y_j$ é o erro observado na saída do sistema para o objeto j.

O processo de treinamento do estimador tem por objetivo corrigir esse erro observado e, para isso, busca minimizar um critério (função objetivo) baseado em e_j, de maneira que os valores de y_j estejam próximos dos de d_j no sentido estatístico. Se a equação de regressão aproxima suficientemente bem os dados de treinamento, então ela pode ser usada para estimar o valor de uma variável (y) a partir do valor da outra variável (X), assumindo uma relação linear entre essas variáveis. Em suma, *a regressão linear procura pelos coeficientes da reta que minimizam a distância dos objetos à reta*. Apesar da sua simplicidade, modelos lineares costumam ser surpreendentemente competitivos.

A regressão linear usa o método dos mínimos quadrados para estimar os seus coeficientes, buscando minimizar a soma dos quadrados dos erros (SQE). No caso de um coeficiente ser zero, a influência da variável de entrada no modelo é removida (pois $0\ x_i = 0$*). O modelo se torna menos complexo e com melhor interpretabilidade, pois as variáveis não relevantes (que não estão realmente associadas à resposta) são eliminadas.

Os **métodos de regularização** são extensões de treinamento do modelo linear que, além de buscar minimizar a soma dos quadrados dos erros (SQE), buscam reduzir a complexidade do modelo através de uma função de penalidade. A **regularização Ridge** busca minimizar também a soma dos quadrados dos coeficientes (conhecida também como regularização L2) e a **regularização Lasso** busca minimizar também a soma do valor absoluto dos coeficientes (conhecida também como regularização L1). Esses métodos são especialmente eficazes quando há correlação entre os atributos de entrada, e os mínimos quadrados comuns superestimam os dados de treinamento, ocorrendo o *overfitting*.

O parâmetro de ajuste λ serve para controlar o impacto da penalidade. Quando $\lambda = 0$, o termo da penalidade não tem efeito e o resultado é similar ao método dos mínimos quadrados da regressão linear. Quanto maior é λ, maior é o impacto da penalidade e maior a diminuição dos coeficientes.

Na regularização Ridge, a penalidade poderá diminuir todos os coeficientes para próximo de zero, mas nunca exatamente zero. O modelo gerado sempre terá todas as variáveis preditoras e não é robusto a *outliers*, podendo prejudicar a sua interpretabilidade, mas sendo capaz de aprender padrões mais complexos. Já na regularização Lasso, a penalidade pode levar alguns coeficientes a exatamente zero (quando λ for suficientemente grande), realizando a seleção de variáveis preditoras e facilitando a interpretabilidade do modelo, que é mais simples e robusto a *outliers*, mas não é capaz de aprender padrões mais complexos.

Regressão logística (classificação)

A **regressão logística**, apesar do nome, é um algoritmo utilizado para problemas de classificação, e seu funcionamento lembra muito o funcionamento do algoritmo de regressão linear. O algoritmo da

regressão logística é usado para estimar valores discretos (valores binários como *0/1*, *sim/não*, *verdadeiro/falso*) com base em um conjunto de variáveis independentes.

Internamente, a regressão logística calcula a probabilidade de ocorrência (*p*) de um evento, ou seja, seus valores de saída estão entre 0 e 1, e essa saída é mapeada na classe correspondente. Assim, a *regressão logística é análoga à regressão linear múltipla* (desejamos modelar *y* como uma função linear de *X*), mas neste caso a saída é uma classe.

Se utilizássemos a fórmula da regressão linear múltipla $y = \beta_0 + \beta 1_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$, não garantiríamos que a saída ficasse entre 0 e 1. Assim, modelamos a saída utilizando uma função logística (também chamada de *sigmoide*), uma curva em forma de *S* que pode mapear qualquer número em um intervalo entre 0 e 1:

$$f(x) = \frac{1}{1+e^{-x}}$$

Figura 10.37: Função logística.

onde f(x) é a saída prevista. Veja o exemplo de uma transformação logística entre -6 e 6:

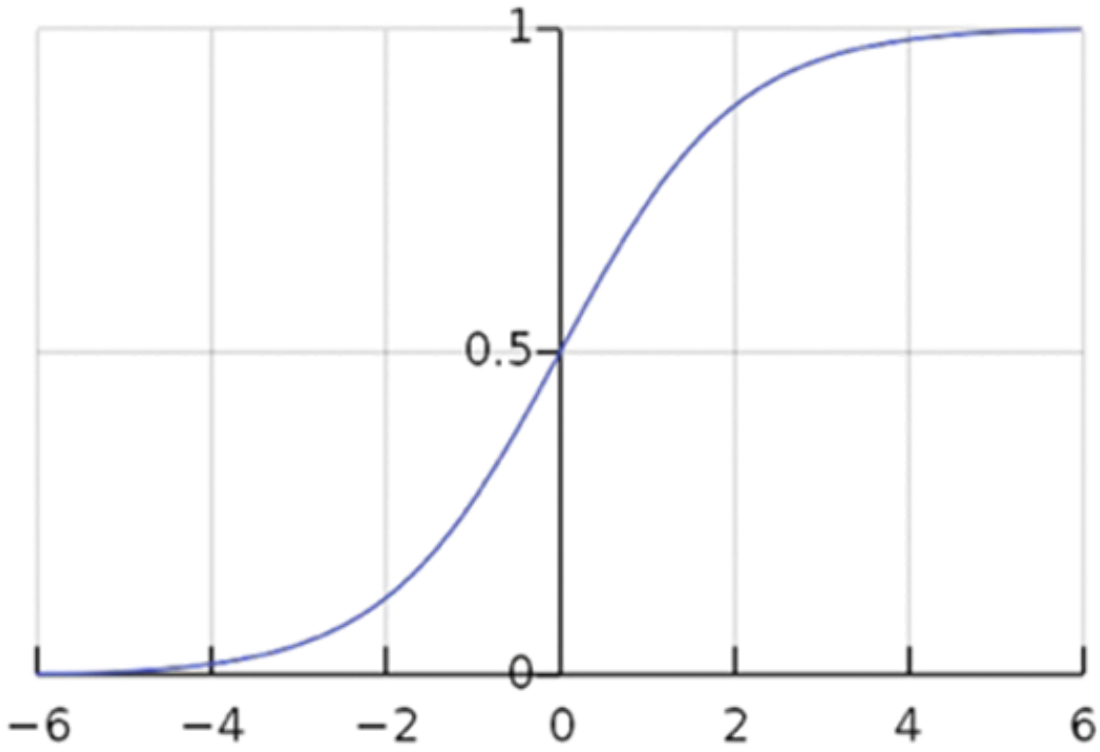


Figura 10.38: Transformação logística entre -6 e 6. Fonte: Wikipedia.

De forma similar à regressão linear, a regressão logística usa uma equação como representação, em que os valores de entrada (*X*) são combinados linearmente usando coeficientes para predizer um valor de saída (*y*):

$$\hat{y} = p = \frac{1}{1 + e^{-(b0 + b1 \times x1)}}$$

Figura 10.39: Regressão logística.

O valor de saída (entre 0 e 1) é então arredondado para um valor binário (0 ou 1) e mapeado na classe correspondente (se for maior que 0,5, a classe é 1, caso contrário, a classe é 0).

Os coeficientes da regressão logística podem ser estimados usando os dados de treinamento, através do método de *estimação de máxima verossimilhança*, um método matemático (cujo detalhamento está fora do escopo deste livro) que busca valores para os coeficientes de forma a minimizar o erro nas probabilidades preditas pelo modelo para os dados. Os melhores coeficientes resultarão em um modelo que vai ter como saída valor muito próximo de 1 para a classe padrão e um valor muito próximo de 0 para a outra classe. Após determinados os coeficientes, para fazer predições com a regressão logística, basta aplicar a equação resultante.

10.4 Exemplos práticos

Você pode acessar os códigos-fontes do exemplo prático de classificação em Python e do exemplo prático de regressão em Python pelos respectivos links a seguir:

https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_classificacao.ipynb

https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_regressao.ipynb

Exemplo prático de classificação em Python

Para ilustrar como aplicamos os algoritmos de *Machine Learning* na prática, vamos examinar o popular *dataset* **Wine** (disponível em https://archive.ics.uci.edu/ml/datasets/Wine), extraído a partir de uma análise química de vinhos cultivados em uma mesma região da Itália, mas derivados de três produtores diferentes. O objetivo desse *dataset* é identificar o produtor do vinho com base em 13 características químicas do vinho, ou seja, é um problema de

classificação. O *dataset* contém 178 instâncias (linhas), sendo 59 do produtor 1, 71 do produtor 2 e 48 do produtor 3.

Vejamos um exemplo de código em Python usando a biblioteca *Scikit-learn*. Primeiramente, vamos carregar o *dataset* e separar em bases de treino e teste através do método *holdout*. Em seguida, para a base de treino, vamos avaliar a acurácia dos modelos treinados com os algoritmos *regressão logística*, KNN, *árvore de classificação*, Naïve Bayes e SVM, utilizando sua configuração padrão da biblioteca *Scikit-learn*, ou seja, sem variar seus hiperparâmetros (exceto na regressão logística, em que utilizaremos um parâmetro para limitar o número de iterações e evitar que o código demore muito tempo para ser executado). Para uma melhor avaliação, utilizaremos o método de validação cruzada *10-fold* e compararemos os resultados graficamente através de *boxplots*.

Todo o código está comentado, para facilitar o entendimento. Iniciaremos esta prática importando os pacotes necessários:

```
# Configuração para não exibir os warnings
import warnings
warnings.filterwarnings("ignore")

# Imports necessários
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine # para importar o dataset wine
from sklearn.model_selection import train_test_split # para particionar em
bases de treino e teste (holdout)
from sklearn.model_selection import KFold # para preparar os folds da
validação cruzada
from sklearn.model_selection import cross_val_score # para executar a
validação cruzada
from sklearn.metrics import accuracy_score # para a exibição da acurácia
do modelo
from sklearn.linear_model import LogisticRegression # algoritmo regressão
logística
```

```
from sklearn.neighbors import KNeighborsClassifier # algoritmo KNN
from sklearn.tree import DecisionTreeClassifier # algoritmo árvore de
classificação
from sklearn.naive_bayes import GaussianNB # algoritmo Naive Bayes
from sklearn.svm import SVC # algoritmo SVM
```

A seguir, vamos preparar os dados. Vamos carregar o *dataset* a partir da biblioteca *Scikit-learn*, aplicar o *holdout* para efetuar a divisão em bases de treino (80%) e teste (20%) e separar em 10 *folds* usando a validação cruzada.

```
# Carga do dataset
wine = load_wine()
dataset = pd.DataFrame(wine.data, columns=wine.feature_names) # conversão
para dataframe
dataset['target'] = wine.target # adição da coluna target

# Separação em bases de treino e teste (holdout)
array = dataset.values
X = array[:,0:13] # atributos
y = array[:,13] # classe (target)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=7) # faz a divisão

# Definindo a métrica de avaliação dos algoritmos
scoring = 'accuracy'

# Criando os folds para a validação cruzada
num_particoes = 10 # número de folds da validação cruzada
kfold = KFold(n_splits=num_particoes, shuffle=True, random_state=7) # faz
o particionamento em 10 folds
```

Em seguida, passaremos para a etapa de modelagem. Definiremos uma *semente global* para esta célula de código (a *semente* é necessária para garantir a reprodutibilidade desse código, com os mesmos resultados) e criaremos os modelos com os algoritmos KNN, árvore de classificação, Naïve Bayes e SVM, adicionando-os em uma lista. Depois, cada um desses modelos será treinado e avaliado com a base de treino, usando a validação cruzada *10-fold*. O resultado

médio da acurácia de cada modelo será impresso, bem como um gráfico *boxplot* sumarizando os resultados das 10 execuções (correspondentes aos 10 *folds*).

```
# Modelagem

# Definindo uma seed global para esta célula de código
np.random.seed(7)

# Listas para armazenar os modelos, os resultados e os nomes dos modelos
models = []
results = []
names = []

# Preparando os modelos e adicionando-os em uma lista
models.append(('LR', LogisticRegression(max_iter=200)))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))

# Avaliando um modelo por vez
for name, model in models:
  cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std()) #
média e desvio padrão dos 10 resultados da validação cruzada
  print(msg)

# Boxplot de comparação dos modelos
fig = plt.figure()
fig.suptitle('Comparação da Acurácia dos Modelos')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Saída:

```
LR: 0.943810 (0.051981)
KNN: 0.675238 (0.087929)
CART: 0.880476 (0.055183)
NB: 0.971429 (0.047380)
SVM: 0.683333 (0.078282)
```

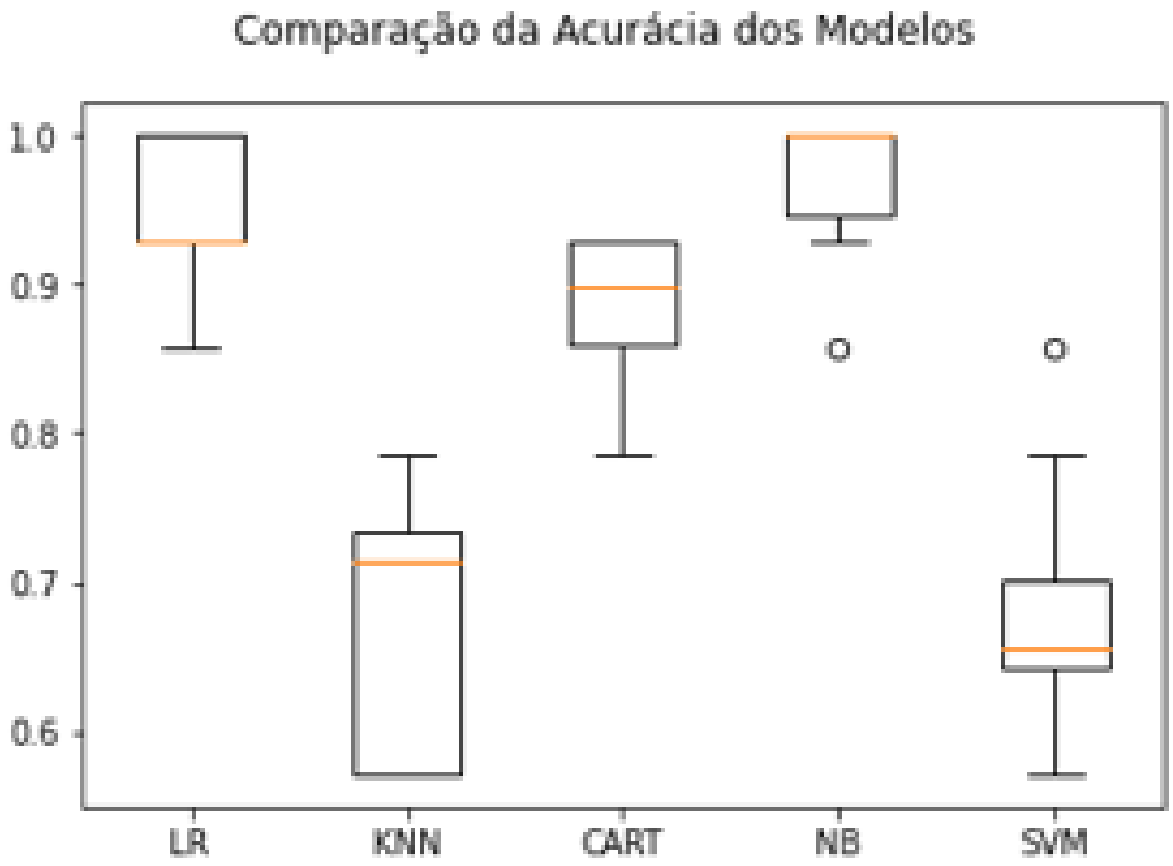


Figura 10.40: Saída.

Analisando os resultados, verificamos que, considerando a acurácia média, o modelo treinado com o Naïve Bayes apresentou os melhores resultados (97% de acurácia média), seguido do modelo treinado com a regressão logística (94% de acurácia média), ambos com desvio padrão equivalente (5%). Já analisando os *boxplots*, vemos que a mediana da acurácia do modelo treinado com o Naïve Bayes é superior à do modelo treinado com a regressão logística,

indicando que possivelmente seguiríamos com o Naïve Bayes como escolha de algoritmo. Neste caso, construiremos um novo modelo, treinado com toda a base de treino. Este modelo será avaliado utilizando a base de teste:

```
# Criando um modelo com todo o conjunto de treino
model = GaussianNB()
model.fit(X_train, y_train)

# Fazendo as predições com o conjunto de teste
predictions = model.predict(X_test)

# Estimando a acurácia no conjunto de teste
print(accuracy_score(y_test, predictions))
```

Saída:

```
1.0
```

Repare que conseguimos obter uma acurácia de teste de 100%. Reforçamos que este resultado não é comum quando trabalhamos com dados de problemas reais, mas possível com *datasets* simplificados similares a este, trabalhados e disponibilizados apenas para fins acadêmicos.

Ressaltamos que esse é um exemplo simples, apenas para ilustrar neste livro a aplicação de algoritmos de *Machine Learning*. Em problemas reais (com dados mais "sujos" e complexos), provavelmente trabalharíamos com algumas operações de pré-processamento de dados (tais como normalização e padronização para dados quantitativos e *One-Hot Encoding* para dados qualitativos nominais) e também experimentaríamos variar os hiperparâmetros dos algoritmos, o que resultaria em um código mais complexo, como veremos nos capítulos a seguir.

Exemplo prático de regressão em Python

Para ilustrar como aplicamos os algoritmos de *Machine Learning* de regressão na prática, vamos examinar o *dataset* Diabetes (para

regressão), disponível em https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html. Esse *dataset* contém 10 variáveis de linha de base sobre 442 pacientes com diabetes: idade, sexo, índice de massa corporal, pressão arterial média e seis medições de soro sanguíneo. A variável *target* é uma medida quantitativa da progressão da doença um ano após a linha de base, ou seja, é um problema de regressão.

Vejamos um exemplo de código em Python usando a biblioteca *Scikit-learn*. Primeiramente, vamos carregar o *dataset* (que está disponível também na biblioteca *Scikit-learn*) e separar em bases de treino e teste através do método *holdout*. Em seguida, para a base de treino, vamos avaliar o MSE e o RMSE dos modelos treinados com os algoritmos *regressão linear*, *regressão linear com regularização Ridge*, *regressão linear com regularização Lasso*, *KNN*, *árvore de regressão* e *SVM*, utilizando sua configuração padrão da biblioteca *Scikit-learn*, ou seja, sem variar seus hiperparâmetros. Para uma melhor avaliação, utilizaremos o método de validação cruzada *10-fold* e compararemos os resultados graficamente através de *boxplots*.

Todo o código está comentado, para facilitar o entendimento. Iniciaremos esta prática importando os pacotes necessários:

```
# Configuração para não exibir os warnings
import warnings
warnings.filterwarnings("ignore")

# Imports necessários
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes # para importar o dataset diabetes
from sklearn.model_selection import train_test_split # para particionar em bases de treino e teste (holdout)
from sklearn.model_selection import KFold # para preparar os folds da validação cruzada
```

```
from sklearn.model_selection import cross_val_score # para executar a
validação cruzada
from sklearn.metrics import mean_squared_error # métrica de avaliação MSE
from sklearn.linear_model import LinearRegression # algoritmo regressão
linear
from sklearn.linear_model import Ridge # algoritmo regularização Ridge
from sklearn.linear_model import Lasso # algoritmo regularização Lasso
from sklearn.neighbors import KNeighborsRegressor # algoritmo KNN
from sklearn.tree import DecisionTreeRegressor # algoritmo árvore de
regressão
from sklearn.svm import SVR # algoritmo SVM
```

A seguir, vamos preparar os dados. Vamos carregar o *dataset* a partir da biblioteca *Scikit-learn*, aplicar o *holdout* para efetuar a divisão em bases de treino (80%) e teste (20%) e separar em 10 *folds* usando a validação cruzada.

```
# Preparação dos dados

diabetes = load_diabetes()
dataset = pd.DataFrame(diabetes.data, columns=diabetes.feature_names) #
conversão para dataframe
dataset['target'] = diabetes.target # adição da coluna target

# Separação em bases de treino e teste (holdout)
array = dataset.values
X = array[:,0:10] # atributos
y = array[:,10] # classe (target)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=7) # faz a divisão

# Criando os folds para a validação cruzada
num_particoes = 10 # número de folds da validação cruzada
kfold = KFold(n_splits=num_particoes, shuffle=True, random_state=7) # faz
o particionamento em 10 folds
```

Em seguida, passaremos para a etapa de modelagem. Definiremos uma *semente global* para esta célula de código (a *semente* é

necessária para garantir a reprodutibilidade deste código, com os mesmos resultados) e criaremos os modelos com os algoritmos regressão linear, regressão linear com regularização Ridge, regressão linear com regularização Lasso, KNN, árvore de regressão e SVM, adicionando-os em uma lista. Depois, cada um destes modelos será treinado e avaliado com a base de treino, usando a validação cruzada *10-fold*. O resultado médio da acurácia de cada modelo será impresso, bem como um gráfico *boxplot* sumarizando os resultados das 10 execuções (correspondentes aos 10 *folds*).

```
# Modelagem

# Definindo uma seed global para esta célula de código
np.random.seed(7)

# Listas para armazenar os modelos, os resultados e os nomes dos modelos
models = []
results = []
names = []

# Preparando os modelos e adicionando-os em uma lista
models.append(('LR', LinearRegression()))
models.append(('Ridge', Ridge()))
models.append(('Lasso', Lasso()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVM', SVR()))

# Avaliando um modelo por vez
for name, model in models:
  cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring='neg_mean_squared_error')
  results.append(cv_results)
  names.append(name)
  # imprime MSE, desvio padrão do MSE e RMSE dos 10 resultados da
validação cruzada
  msg = "%s: MSE %0.2f (%0.2f) - RMSE %0.2f" % (name,
abs(cv_results.mean()), cv_results.std(), np.sqrt(abs(cv_results.mean())))
  print(msg)
```

```
# Boxplot de comparação dos modelos
fig = plt.figure()
fig.suptitle('Comparação do MSE dos Modelos')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Saída:

```
LR: MSE 3066.48 (612.06) - RMSE 55.38
Ridge: MSE 3566.43 (805.55) - RMSE 59.72
Lasso: MSE 3948.90 (891.00) - RMSE 62.84
KNN: MSE 3522.14 (721.76) - RMSE 59.35
CART: MSE 6431.26 (1584.05) - RMSE 80.20
SVM: MSE 5285.08 (1186.19) - RMSE 72.70
```

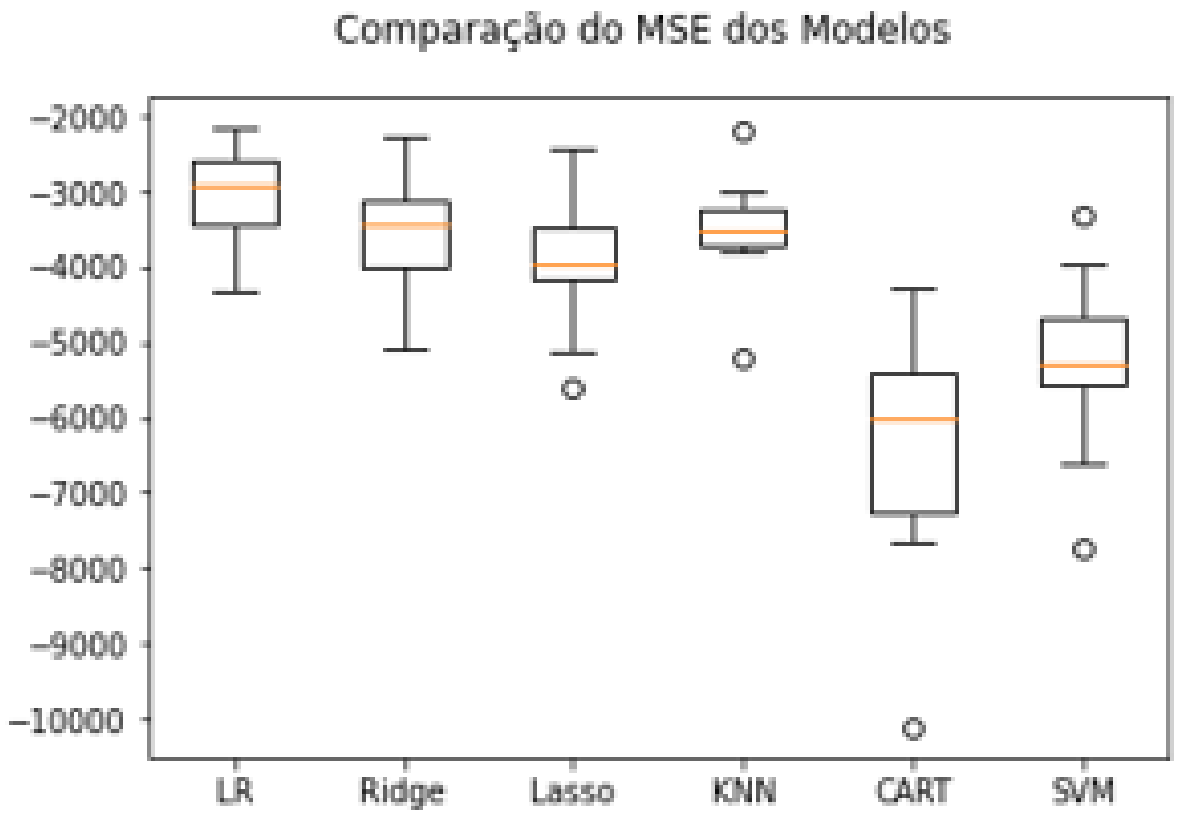


Figura 10.41: Saída.

Analisando os resultados, verificamos que, considerando o MSE (e, consequentemente, o RMSE), o modelo treinado com a árvore de regressão apresentou os melhores resultados (valores de erro menores), indicando que possivelmente seguiríamos com a árvore de regressão como escolha de algoritmo. Neste caso, construiremos um novo modelo, treinado com toda a base de treino. Esse modelo será avaliado utilizando a base de teste:

```
# Criando um modelo com todo o conjunto de treino
model = DecisionTreeRegressor()
model.fit(X_train, y_train)

# Fazendo as predições com o conjunto de teste
predictions = model.predict(X_test)

# Estimando o MSE e o RMSE no conjunto de teste
mse = mean_squared_error(y_test, predictions)
print("MSE %0.2f" % mse)
print("RMSE %0.2f" % np.sqrt(abs(mse)))
```

Saída:

```
MSE 4512.10
RMSE 67.17
```

Assim como no exemplo do problema de classificação, ressaltamos que esse é um exemplo simples, apenas para ilustrar a aplicação de algoritmos de *Machine Learning* em problemas de regressão. Em problemas reais (com dados mais "sujos" e complexos), provavelmente trabalharíamos com algumas operações de pré-processamento de dados (tais como normalização e padronização para dados quantitativos, e *One-Hot-Encoding* para dados qualitativos nominais) e também experimentaríamos variar os hiperparâmetros dos algoritmos, o que resultaria em um código mais complexo, como veremos nos próximos capítulos.

CAPÍTULO 11
Recursos avançados de Machine Learning

11.1 Ensembles

Os **métodos *ensemble*** (ou comitês) combinam vários modelos de *Machine Learning*, o que pode fazer com que os resultados sejam melhores do que quando usamos apenas um modelo. Muitas vezes, esses modelos têm melhor desempenho preditivo em comparação com um único modelo, sendo os primeiros colocados em muitas competições de aprendizado de máquina de prestígio, como as do *Kaggle*.

Conceitualmente, *ensembles* são meta-algoritmos que combinam várias técnicas de aprendizado de máquina em um modelo preditivo, com o objetivo de diminuir a variância, o viés ou melhorar as predições, e podem ser de dois grupos:

- **Métodos sequenciais**, em que os modelos-base são gerados sequencialmente e exploram a dependência entre os modelos de base. O desempenho geral pode ser aprimorado ponderando os exemplos previamente rotulados incorretamente com maior peso;
- **Métodos paralelos**, em que os modelos-base são gerados em paralelo e exploram a independência entre os modelos-base e a predição final é uma consolidação da predição dos modelos base.

A maioria dos métodos *ensemble* usa um único algoritmo básico, produzindo *ensembles* **homogêneos**, mas também há métodos que usam algoritmos diferentes, os *ensembles* **heterogêneos**. Para que o *ensemble* seja melhor do que qualquer um de seus modelos individuais, os modelos-base precisam ter boa precisão e serem diversificados, idealmente, com alta variância e baixo viés (*bias*).

Os métodos *ensemble* mais populares são:

- ***Voting***: constrói vários modelos (geralmente de tipos de algoritmos diferentes) e calcula estatísticas simples (ex.: moda ou média) para combinar as predições;
- ***Bagging***: constrói vários modelos (geralmente do mesmo tipo de algoritmo) a partir de diferentes subamostras do conjunto de dados de treinamento;
- ***Boosting***: constrói vários modelos (geralmente do mesmo tipo de algoritmo) e cada um deles aprende a corrigir os erros de predição de um modelo anterior na sequência de modelos.

Vamos então estudar com mais detalhes cada um desses métodos.

Voting

Os *ensembles* do tipo *voting* utilizam uma das técnicas mais simples de combinar predições de vários modelos. Criam dois ou mais modelos independentes a partir de todo o conjunto de dados de treinamento. A predição final geralmente é feita por votação majoritária (para problemas de classificação) ou média (para problemas de regressão).

No *ensemble* do tipo *voting*, as predições dos submodelos podem ser ponderadas, mas é difícil especificar os pesos dos classificadores manualmente ou mesmo heuristicamente. Existem métodos mais avançados que podem aprender a ponderar as predições dos submodelos construindo um novo modelo. Essa técnica é conhecida como *Stacking (Stacked Aggregation)*.

Em problemas de classificação, a classe predita pelo *ensemble* é a mais votada entre os classificadores. O *ensemble* geralmente tem melhor acurácia que o melhor classificador do *ensemble*, e mesmo que todos os classificadores sejam "fracos" (um pouco melhores que escolha aleatória), o *ensemble* ainda pode ser um classificador "forte", desde que os classificadores sejam numerosos e diversos o suficiente.

A figura a seguir ilustra a decisão de um *ensemble* para problemas de classificação. Repare que três dos quatro modelos-base tiveram como saída a classe 1, apenas um deles teve como saída a classe 2. Como a classe 1 teve três dos quatro votos, ela é a resposta do *ensemble* para o problema.

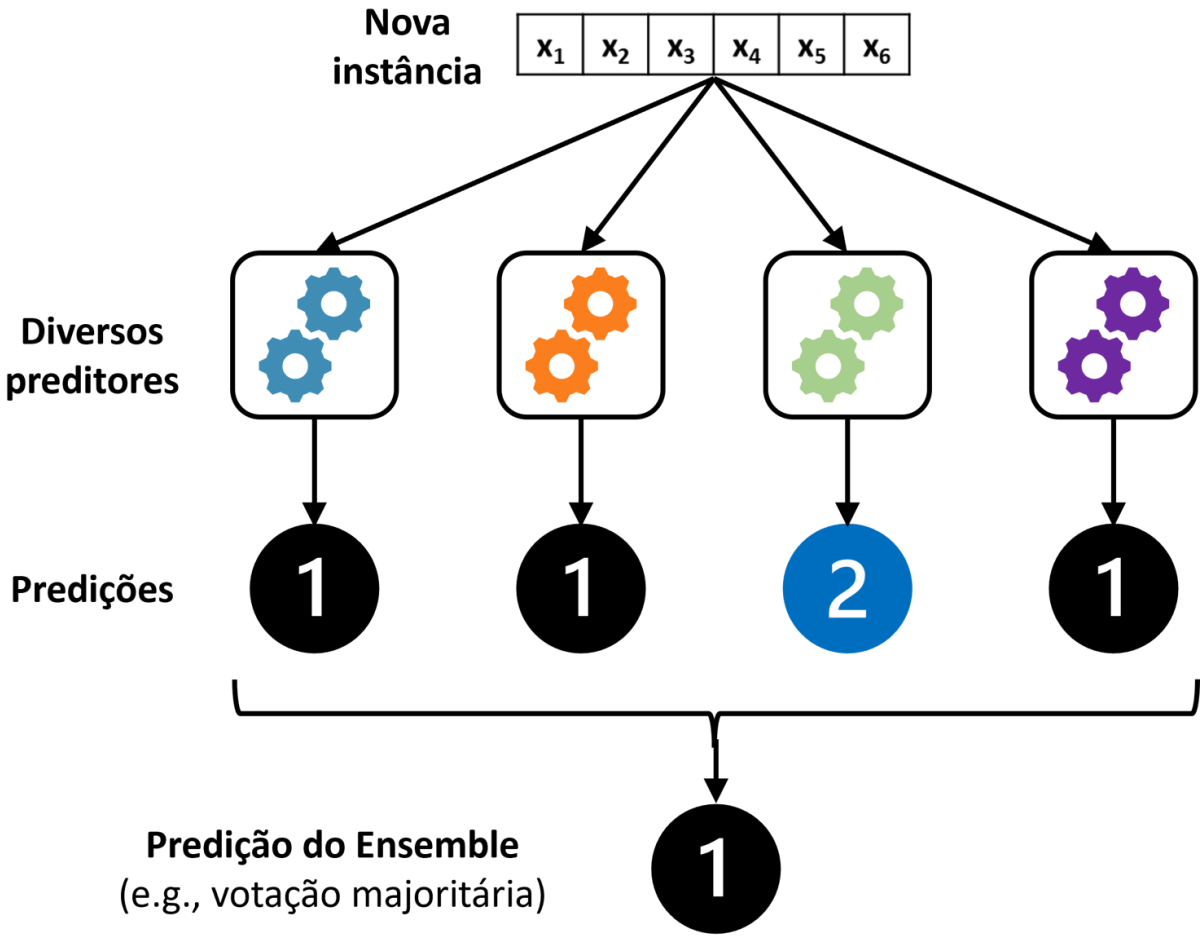


Figura 11.1: Exemplo de ensemble do tipo voting. Fonte: Adaptada de Géron, 2022.

Recomenda-se usar *ensembles* do tipo *voting* quando todos os modelos do *ensemble* tiverem, em geral, boa performance, e quando a maioria dos modelos concordar entre si. Se qualquer modelo-base do *ensemble* tiver um desempenho melhor que o *ensemble* em si, esse modelo provavelmente deverá ser usado em vez do *ensemble*. Também podemos usar *voting ensembles* para

combinar diferentes versões de um mesmo algoritmo de *Machine Learning*, variando os seus hiperparâmetros.

Bagging

Em *ensembles* do tipo ***bagging***, são construídos vários modelos (geralmente do mesmo tipo de algoritmo) a partir de diferentes subamostras do conjunto de dados de treinamento. Para tal, é utilizada a técnica de ***bootstrap***, um método estatístico poderoso para estimar uma quantidade (como a média ou o desvio padrão) a partir de uma amostra pequena de dados.

Vejamos um exemplo. Imagine que temos uma amostra de 100 valores (x_1 a x_{100}) e gostaríamos de obter uma estimativa da média dessa amostra. Sabemos que é possível calcular a média diretamente da amostra como:

$$média(x) = \frac{1}{100} \times \sum_{i=1}^{100} x_i$$

Figura 11.2: Média da amostra.

Entretanto, como sabemos que a amostra é pequena, essa média provavelmente tem um erro. Podemos melhorar a estimativa de nossa média usando o *bootstrap*. Para isso:

- Crie muitas subamostras aleatórias (por exemplo, 1000) do conjunto de dados com substituição;
- Calcule a média de cada subamostra;
- Calcule a média das médias coletadas e use-a como a média estimada para os dados.

Usando esse racional, os *ensembles* do tipo *bagging* utilizam o mesmo algoritmo para cada modelo base, treinando cada um deles em um subconjunto aleatório do conjunto de treino. Em geral, esse tipo de *ensemble* trabalha com árvores de decisão como algoritmo dos seus modelos base. Após o treinamento dos modelos base, a predição do *ensemble* para uma nova instância é a agregação das predições individuais, geralmente pela classe mais votada (para problemas de classificação) ou média das saídas (para problemas de regressão).

A figura a seguir ilustra o treinamento de um *ensemble* do tipo *bagging* para problemas de classificação. Repare que, a partir do conjunto de treino, foram construídas quatro subamostras (usando *bootstrap*, ou seja, com reposição) e cada uma delas foi utilizada para treinar um modelo-base distinto.

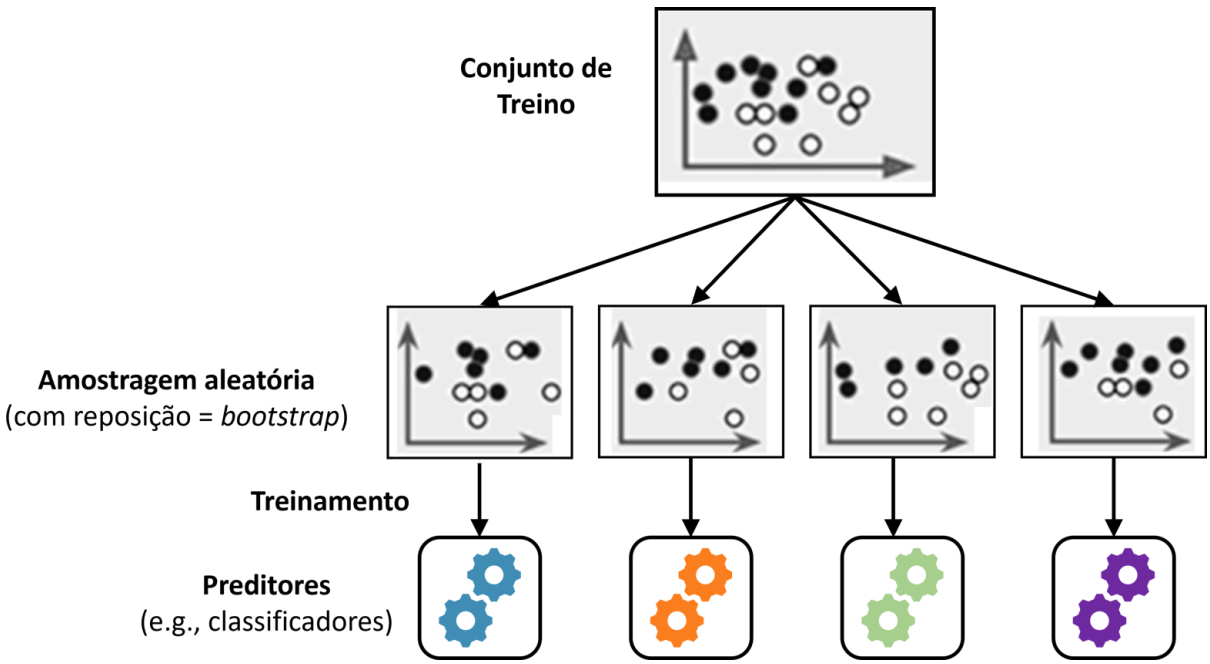


Figura 11.3: Exemplo de ensemble do tipo bagging. Fonte: Adaptada de Géron, 2022.

Uma forma de reduzir a variância de uma predição é calcular a média (ou moda) de várias predições. Os *ensembles* do tipo *bagging* fazem isso, pois consolidam as predições de várias subamostras do conjunto de treinamento. Assim, esses *ensembles* podem ser usados

para reduzir a variância dos modelos-base com alta variância (por exemplo, árvores de decisão, que são muito sensíveis aos dados específicos que são treinados). Como utiliza apenas um subamostra, cada modelo-base tem maior *bias* do que se fosse treinado com todo o conjunto de treinamento, mas a agregação reduz tanto o *bias* quanto a variância do *ensemble*.

O funcionamento dos *ensembles* do tipo *bagging* pode ser sumarizado por:

1. Crie várias subamostras (cada uma com aproximadamente 60% do *dataset* de treinamento original) aleatórias do conjunto de dados de treino, com reposição;
2. Escolha um algoritmo (por exemplo, árvore de decisão) e treine um modelo de *Machine Learning* usando cada subamostra;
3. Dada uma nova instância de dados, calcule a predição do *ensemble* (para problemas de classificação, a classe mais frequente; para problemas de regressão, a média das saídas de cada modelo individual).

O número de subamostras a serem geradas e, consequentemente, o número de modelos-base a serem criados é um parâmetro do *ensemble*, que pode ser escolhido experimentalmente.

Um exemplo muito conhecido dos *ensembles* do tipo *bagging* é o algoritmo ***Random Forest*** (ou floresta aleatória), que representa uma melhoria em relação ao *bagging* tradicional com árvores de decisão. Esse algoritmo também utiliza diferentes subamostras aleatórias do conjunto de treinamento para a criação de modelos base. Além disso, durante a etapa de treinamento de cada um desses modelos, é utilizado um subconjunto aleatório dos atributos para decidir como fazer a divisão da árvore em cada nó, procurando o melhor atributo dentre a subamostra de atributos. Para esta amostra de atributos, o algoritmo busca o ponto de corte ótimo para cada atributo. Isso resulta em uma maior diversidade de árvores, geralmente produzindo um *ensemble* melhor.

Outro conhecido exemplo de *ensemble* do tipo *bagging* é o algoritmo ***Extra Trees***, também conhecido como *Extremely Randomized Trees* (árvores extremamente aleatórias), que é uma extensão do *Random Forest*. Ao contrário do *bagging* e do *Random Forest* (que constroem cada árvore de decisão a partir de uma amostra *bootstrap* do conjunto de dados de treinamento), o *Extra Trees* ajusta cada árvore de decisão utilizando todo o conjunto de dados de treinamento. Para o treinamento de cada árvore de decisão base, o *Extra Trees* utiliza para cada nó da árvore pontos de corte aleatórios, considerando um subconjunto também aleatório de atributos. Essas divisões são avaliadas e a melhor delas é a escolhida para o nó. Devido a essa grande aleatoriedade, o *Extra Trees* é um modelo mais rápido computacionalmente do que o *Random Forest*, pois não é necessário avaliar onde realizar cada divisão.

Boosting

Os *ensembles* do tipo ***boosting*** são *ensembles* sequenciais. Eles criam uma sequência de modelos na qual um modelo tenta corrigir os erros do modelo anterior. Esse tipo de *ensemble* busca converter modelos fracos em modelos fortes e é do tipo homogêneo, sendo geralmente utilizado com árvores de decisão.

O ***Adaboost*** foi o primeiro algoritmo de *boosting* bem-sucedido e é o mais popular. Esse algoritmo pondera as instâncias no conjunto de dados de acordo com a facilidade ou dificuldade de classificá-las no modelo corrente, o que permite prestar mais atenção nos exemplos "mais difíceis" na construção de modelos subsequentes. Para isso, um maior peso é dado aos exemplos que foram classificados incorretamente no modelo anterior.

O funcionamento do *Adaboost* para problemas de classificação pode ser sumarizado por:

1. Treine um primeiro modelo no qual cada observação recebe um peso igual;

2. Avalie o primeiro modelo e atualize os pesos das observações, aumentando os pesos das observações difíceis de classificar e diminuindo os pesos para as fáceis de classificar;
3. Calcule o peso do modelo de acordo com a sua avaliação de desempenho (quanto menor seu erro, maior seu peso);
4. Treine um segundo modelo usando os dados anteriores ponderados, com o objetivo de melhorar as predições do modelo anterior;
5. Repita os passos 2, 3 e 4 para um número especificado de iterações.

A predição final do *Adaboost* é calculada através da ponderação das saídas de cada modelo-base pelo seu peso, e a classe predita é aquela que recebe a maioria dos votos ponderados.

A figura a seguir ilustra o treinamento de um *ensemble* do tipo *boosting* que utiliza o algoritmo *Adaboost* para problemas de regressão. Repare que inicialmente um modelo é treinado e utilizado para fazer predições no conjunto de treinamento. Em seguida, o peso relativo das instâncias de treinamento que tiveram a predição incorreta é aumentado. Um segundo modelo é treinado usando os pesos atualizados e as predições são novamente realizadas no conjunto de treinamento. Os pesos das instâncias são atualizados e assim sucessivamente.

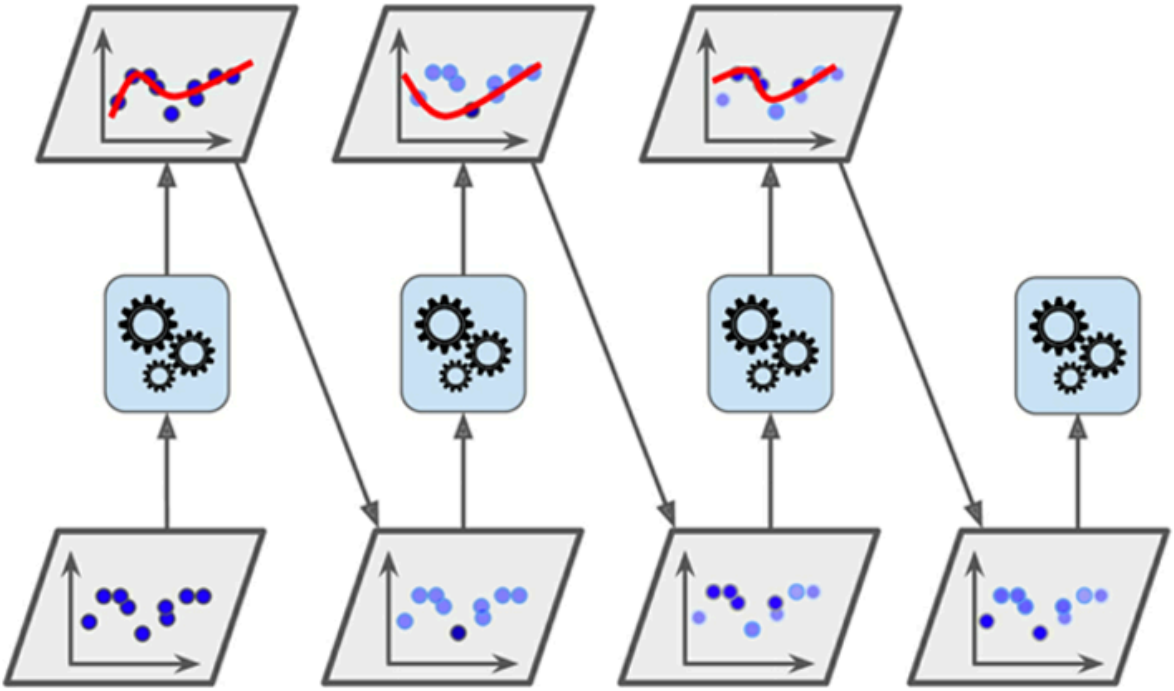


Figura 11.4: Exemplo de ensemble do tipo boosting. Fonte: Géron, 2022.

Outro algoritmo de *boosting* muito popular é o ***Gradient Boosting***, que serve de base para a implementação de sua versão otimizada, conhecida como *XGBoost (Extreme Gradient Boosting)*. O *Gradient Boosting* é uma variação do *Adaboost* que tenta ajustar o modelo-base novo aos erros residuais cometidos pelo modelo-base anterior, em vez de ajustar os pesos das observações a cada iteração. Para isso, o *Gradient Boosting* usa o procedimento de descida do gradiente (baseado em erros residuais, método cujo detalhamento está fora do escopo deste livro) para adicionar novos modelos-base, e o novo modelo reduz a perda, indo na direção do gradiente. É uma das técnicas de *ensemble* mais sofisticadas e uma das melhores técnicas disponíveis para melhorar o desempenho via *ensembles*.

11.2 Feature selection

Até agora, discutimos diversos algoritmos de *Machine Learning* aplicados a problemas de classificação e regressão, e vimos que os *ensembles* são algoritmos que combinam diversos modelos com o objetivo de termos resultados melhores. Entretanto, os atributos (*X*) dos dados usados para treinamento dos modelos de *Machine Learning* têm uma enorme influência no seu desempenho. Atributos irrelevantes ou parcialmente relevantes podem afetar negativamente o resultado do modelo, especialmente em algoritmos lineares como regressão linear e regressão logística.

Assim, quando trabalhamos em problemas de *Machine Learning*, é importante prestarmos atenção no processo de seleção de atributos (características), conhecido como ***feature selection***. Esse processo antecede a etapa de *modelagem e inferência* do esquema básico de um projeto de Ciência de Dados (visto anteriormente neste livro) e consiste em selecionar os atributos (*X*) que mais contribuem para a saída (*y*) e utilizá-los para treinamento do modelo, ou seja, reduzir o número de variáveis de entrada do modelo.

Dentre as possíveis vantagens de realizar a etapa de *feature selection*, podemos citar:

1. A **redução de *overfitting***, pois, com dados menos redundantes, temos um menor risco de tomar decisões com base em ruído;
2. A **melhoria do resultado do modelo**, pois dados menos "enganosos" e mais relevantes para o problema trazem uma melhor precisão do modelo;
3. **Redução do tempo de treinamento**, pois quando trabalhamos com menos dados temos um treinamento mais rápido; e
4. **Melhoria da interpretabilidade do modelo**, pois com menos atributos há mais chances de um melhor entendimento do processo que gerou os dados e do resultado gerado pelo modelo.

Neste ponto, é importante destacarmos a distinção entre *feature selection* e *redução de dimensionalidade*. Ambos os métodos tendem a reduzir o número de atributos (colunas) no conjunto de dados. Porém, os métodos de redução de dimensionalidade criam novas combinações dos atributos (também conhecido como *Feature Transformation*) usando operações de álgebra linear, enquanto os métodos de seleção de atributos excluem atributos do *dataset* (*feature selection*). São exemplos de métodos de redução de dimensionalidade o **PCA** (*Principal Component Analysis*) e o **SVD** (*Singular Value Decomposition*), cujo detalhamento está fora do escopo deste livro.

Os métodos de *feature selection* podem ser **não supervisionados**, quando não utilizam a variável de saída (*y*), por exemplo, removendo atributos duplicados ou redundantes na etapa de pré-processamento de dados; ou **supervisionados**, quando utilizam a variável de saída (*y*), por exemplo, removendo atributos irrelevantes para o problema, o que é possível fazer através de técnicas distintas.

Na técnica **seleção univariada**, testes estatísticos são usados para selecionar os atributos que tenham relação mais forte com a variável de saída. Na técnica **eliminação recursiva de atributos**, os atributos são removidos recursivamente e vai sendo construído um modelo com os que permanecem, usando o resultado do modelo (por exemplo, a acurácia) para identificar quais atributos mais contribuem para a predição da variável de saída. Já na técnica **importância de atributos**, são utilizados algoritmos de *Machine Learning* (como *Random Forest* ou *Extra Trees*) para estimar a importância dos atributos, fazendo uma pré-seleção.

É possível também realizar a operação de *feature selection* de forma intrínseca, através de algoritmos que fazem a seleção de atributos automaticamente como parte do treinamento do modelo (por exemplo, regressão linear com regularização Lasso e árvores de decisão).

É importante ressaltar que, assim como não há um "melhor" algoritmo de *Machine Learning*, não existe uma "melhor" técnica de *feature selection*. É preciso avaliar experimentalmente a técnica que produz os melhores resultados para cada problema específico.

11.3 Pipelines

As operações de *feature selection* e transformação de dados fazem parte da etapa de pré-processamento de dados do esquema básico de um projeto de Ciência de Dados. Essa costuma ser a etapa mais demorada e trabalhosa do projeto, e consiste na preparação dos dados brutos em um formato apropriado para a modelagem. Na etapa de pré-processamento de dados, pode ser necessário, por exemplo, tratar dados faltantes e dados discrepantes (*outliers*); transformar dados e selecionar as características mais adequadas para construir os modelos na etapa seguinte, de modelagem e inferência.

Realizar a preparação de dados é uma tarefa que deve ser feita com atenção e cuidado, de forma a evitar um problema conhecido como *Data Leakage* (vazamento de dados). Esse problema ocorre quando há "vazamento" dos dados da base de teste para a base de treino, o que pode resultar em uma estimativa incorreta do desempenho do modelo ao realizar predições nos novos dados. Ou seja, é como se na fase de treinamento do modelo fossem reveladas a ele, de forma antecipada e indevida, as informações "futuras" da base de teste.

Preparar os dados usando normalização em todo o *dataset* antes do treinamento do modelo não seria recomendado, porque, para realizar essa transformação, calcularíamos os valores mínimo e máximo de todo o *dataset*, ou seja, o conjunto de treino normalizado seria influenciado pelo intervalo dos dados do conjunto de teste, uma vez que o conjunto de treino teria sido ajustado usando os valores mínimo e máximo globais. Apesar de esse

vazamento ser geralmente sutil, ele pode interferir nos resultados do modelo.

Esse problema pode acontecer tanto quando avaliamos nossos modelos com a técnica de *holdout* (dividindo o *dataset* em conjuntos de treino e teste) como quando utilizamos a validação cruzada, pois ambas as técnicas consistem em treinar o modelo com uma parte dos dados e testá-lo com uma parte diferente dos dados, que não deve ser utilizada para o treinamento do modelo.

Para evitar a ocorrência de *Data Leakage*, precisamos garantir uma forte separação dos conjuntos de treino e teste, incluindo na etapa de preparação de dados. Assim, os parâmetros utilizados para a preparação de dados devem ser extraídos apenas do conjunto de treino, aplicados no conjunto de treino e, uma vez ocorrido o treinamento do modelo, a preparação de dados do conjunto de teste deve utilizar os mesmos parâmetros utilizados no conjunto de treino. Para tal, deve-se:

1. Realizar a separação do *dataset* em conjuntos de treino e teste;
2. Obter os parâmetros para a preparação de dados a partir do conjunto de treino;
3. Aplicar a preparação de dados separadamente nos conjuntos de treino e teste, utilizando os parâmetros obtidos no conjunto de treino;
4. Treinar o modelo com os dados de treino e testá-lo com os dados de teste.

Para nos ajudar nessa tarefa e evitar a ocorrência de *Data Leakage*, a biblioteca *Scikit-learn* fornece um recurso conhecido como ***pipeline*** para ajudar a automatizar os fluxos de trabalho de *Machine Learning*. De forma resumida, os *pipelines* permitem que uma sequência linear de operações de preparação de dados seja encadeada com o treinamento do modelo, para que ele seja avaliado em seguida. O objetivo é garantir que todas as etapas do *pipeline* sejam restritas ao conjunto de dados apropriado (como o conjunto de treino do *holdout* ou cada *fold* do procedimento de validação

cruzada), permitindo obter uma estimativa justa do desempenho do modelo com dados não vistos.

De forma mais específica, todo o *pipeline* de modelagem (que inclui a preparação de dados e o treinamento do modelo) deve ser realizado apenas no conjunto de treinamento para evitar o *Data Leakage*. Para avaliar o desempenho desse modelo, as etapas utilizadas no *pipeline* de treino devem ser replicadas para o conjunto de teste, mas com os parâmetros obtidos com o conjunto de treino.

11.4 Otimização de hiperparâmetros

Após uma primeira avaliação, treinando modelos com os algoritmos de *Machine Learning* possíveis para tratar um problema e analisando os resultados obtidos, é interessante realizarmos o *tuning* (ajuste) dos melhores algoritmos encontrados até então, a fim de buscarmos construir o melhor modelo para o problema. Para isso, os algoritmos de *Machine Learning* podem ser parametrizados com os chamados **hiperparâmetros**, a fim de que seu comportamento possa ser ajustado para um determinado problema. Os algoritmos podem ter diversos hiperparâmetros, que podem receber diversos valores para construir o modelo, e muitas vezes encontrar a melhor combinação de hiperparâmetros manualmente pode ser uma tarefa difícil.

Alguns exemplos de hiperparâmetros para os algoritmos que estudamos são:

- *KNN*: número de vizinhos a considerar (*k*) e métrica de distância utilizada;
- *Árvore de decisão*: profundidade máxima da árvore e critério de avaliação da qualidade dos particionamentos;
- *SVM*: tipo de *kernel* utilizado e rigidez da margem;
- *Regressão linear Ridge e Lasso*: impacto da penalidade da regularização;

- *Bagging*: algoritmo dos modelos-base e número de membros do *ensemble*.

Para nos ajudar na tarefa da otimização de hiperparâmetros, a biblioteca *Scikit-learn* fornece uma funcionalidade conhecida como ***grid search*** que nos permite informar em um *grid* quais hiperparâmetros queremos variar, bem como o conjunto de valores que queremos avaliar para cada um deles. Para cada combinação de hiperparâmetros especificada no *grid*, a biblioteca vai sistematicamente construir e avaliar um modelo utilizando validação cruzada, o que nos permitirá escolher ao final desse processo a melhor combinação de hiperparâmetros para o algoritmo em questão.

Finalmente, poderemos preparar o modelo escolhido para ser implantado, treinando-o com todo o *dataset* disponível.

11.5 Exemplos práticos

Você pode acessar o código-fonte do exemplo prático de classificação em Python que utiliza alguns dos recursos avançados que estudamos neste capítulo pelo seguinte link: https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_classificacao_avancada.ipynb.

Você também pode acessar o código-fonte do exemplo prático de algumas técnicas de *feature selection* em Python, em: https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_feature_selection.ipynb.

Exemplo prático de classificação em Python com recursos avançados

Para ilustrar na prática os conceitos que estudamos neste capítulo, vamos ver um exemplo de código. Para esta prática, vamos utilizar o *dataset* ***Pima Indians Diabetes***, que é originalmente do Instituto Nacional de Diabetes e Doenças Digestivas e Renais. Seu objetivo é prever se um paciente terá ou não diabetes, com base em certas medidas de diagnóstico médico. Esse é um subconjunto de um *dataset* maior e, aqui, todos os pacientes são mulheres com pelo menos 21 anos de idade e de herança indígena Pima.

Este *dataset* consiste em vários atributos de exames médicos e uma variável de classe. Vejamos no que consistem os atributos deste *dataset*:

- *Pregnancies*: números de vezes que a mulher já engravidou;
- *Glucose*: concentração de glucose no plasma;
- *BloodPressure*: pressão arterial diastólica;
- *SkinThickness*: dobra da pele do tríceps em milímetros;
- *Insulin*: taxa de insulina;
- *BMI*: índice de massa corpórea (IMC);
- *DiabetesPedigreeFunction*: probabilidade de diabetes com base no histórico familiar;
- *Age*: idade;
- *Outcome*: classe de saída, sendo 0 (sem diabetes) ou 1 (com diabetes).

Todo o código está comentado, para facilitar o entendimento. Vamos iniciar executando um código para que os _warnings não sejam exibidos e, em seguida, vamos importar os pacotes que utilizaremos:

```
# configuração para não exibir os warnings
import warnings
warnings.filterwarnings("ignore")

# Imports necessários
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

Agora, vamos carregar o *dataset* a partir de um arquivo disponível on-line, e exibir as primeiras linhas do *dataset*:

```
# Informa a URL de importação do dataset
url =
"https://raw.githubusercontent.com/profkalinowski/livroescd/main/diabetes.
csv"

# Faz a leitura do arquivo
dataset = pd.read_csv(url, delimiter=',')

# Mostra as primeiras linhas do dataset
dataset.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Figura 11.5: Saída.

Em seguida, vamos realizar a separação do *dataset* nas bases de treino e teste, com o método *holdout*:

```
test_size = 0.20 # tamanho do conjunto de teste
seed = 7 # semente aleatória

# Separação em conjuntos de treino e teste
array = dataset.values
X = array[:,0:8]
y = array[:,8]
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=test_size, shuffle=True, random_state=seed, stratify=y) #
holdout com estratificação

# Parâmetros e partições da validação cruzada
scoring = 'accuracy'
num_particoes = 10
kfold = StratifiedKFold(n_splits=num_particoes, shuffle=True,
random_state=seed) # validação cruzada com estratificação
```

Iniciaremos agora a etapa de *modelagem e inferência*, mas não sabemos de antemão quais algoritmos performarão bem neste conjunto de dados. Assim, usaremos a validação cruzada para treinar e avaliar os modelos usando a métrica *acurácia*. Primeiramente, vamos avaliar os algoritmos com a configuração padrão de hiperparâmetros do *Scikit-learn*:

```
np.random.seed(7) # definindo uma semente global

# Lista que armazenará os modelos
```

```
models = []

# Criando os modelos e adicionando-os na lista de modelos
models.append(('LR', LogisticRegression(max_iter=200)))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))

# Definindo os parâmetros do classificador-base para o BaggingClassifier
base = DecisionTreeClassifier()
num_trees = 100
max_features = 3

# Criando os modelos para o VotingClassifier
bases = []
model1 = LogisticRegression(max_iter=200)
bases.append(('logistic', model1))
model2 = DecisionTreeClassifier()
bases.append(('cart', model2))
model3 = SVC()
bases.append(('svm', model3))

# Criando os ensembles e adicionando-os na lista de modelos
models.append(('Bagging', BaggingClassifier(base_estimator=base,
n_estimators=num_trees)))
models.append(('RF', RandomForestClassifier(n_estimators=num_trees,
max_features=max_features)))
models.append(('ET', ExtraTreesClassifier(n_estimators=num_trees,
max_features=max_features)))
models.append(('Ada', AdaBoostClassifier(n_estimators=num_trees)))
models.append(('GB', GradientBoostingClassifier(n_estimators=num_trees)))
models.append(('Voting', VotingClassifier(bases)))

# Listas para armazenar os resultados
results = []
names = []

# Avaliação dos modelos
```

```
for name, model in models:
    cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)

# Boxplot de comparação dos modelos
fig = plt.figure(figsize=(15,10))
fig.suptitle('Comparação dos Modelos')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Saída:

```
LR: 0.762427 (0.057485)
KNN: 0.711978 (0.058934)
CART: 0.702010 (0.063448)
NB: 0.746060 (0.051145)
SVM: 0.749418 (0.040958)
Bagging: 0.760814 (0.046503)
RF: 0.772263 (0.048493)
ET: 0.756002 (0.066055)
Ada: 0.749418 (0.052763)
GB: 0.750952 (0.057316)
Voting: 0.762507 (0.061246)
```

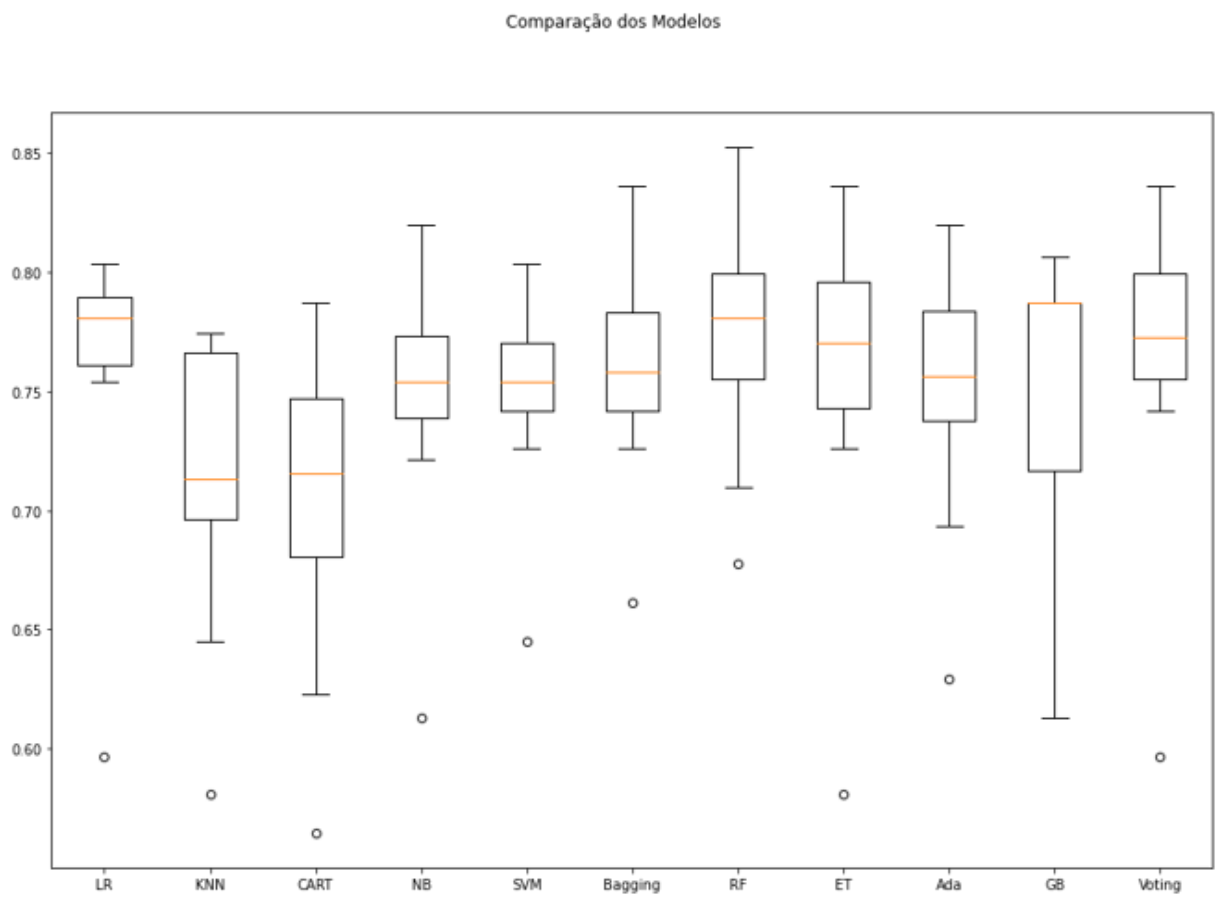


Figura 11.6: Saída.

Em seguida, vamos repetir esse processo, utilizando a biblioteca *Pipeline* para criar e avaliar os modelos através da validação cruzada com os dados padronizados e normalizados (evitando o *Data Leakage*) e comparar o seu resultado com os modelos treinados com o *dataset* original:

```
np.random.seed(7) # definindo uma semente global para este bloco

# Listas para armazenar os pipelines e os resultados para todas as visões
do dataset
pipelines = []
results = []
names = []

# Criando os elementos do pipeline
```

```
# Algoritmos que serão utilizados
reg_log = ('LR', LogisticRegression(max_iter=200))
knn = ('KNN', KNeighborsClassifier())
cart = ('CART', DecisionTreeClassifier())
naive_bayes = ('NB', GaussianNB())
svm = ('SVM', SVC())
bagging = ('Bag', BaggingClassifier(base_estimator=base,
n_estimators=num_trees))
random_forest = ('RF', RandomForestClassifier(n_estimators=num_trees,
max_features=max_features))
extra_trees = ('ET', ExtraTreesClassifier(n_estimators=num_trees,
max_features=max_features))
adaboost = ('Ada', AdaBoostClassifier(n_estimators=num_trees))
gradient_boosting = ('GB',
GradientBoostingClassifier(n_estimators=num_trees))
voting = ('Voting', VotingClassifier(bases))

# Transformações que serão utilizadas
standard_scaler = ('StandardScaler', StandardScaler())
min_max_scaler = ('MinMaxScaler', MinMaxScaler())

# Montando os pipelines

# Dataset original
pipelines.append(('LR-orig', Pipeline([reg_log])))
pipelines.append(('KNN-orig', Pipeline([knn])))
pipelines.append(('CART-orig', Pipeline([cart])))
pipelines.append(('NB-orig', Pipeline([naive_bayes])))
pipelines.append(('SVM-orig', Pipeline([svm])))
pipelines.append(('Bag-orig', Pipeline([bagging])))
pipelines.append(('RF-orig', Pipeline([random_forest])))
pipelines.append(('ET-orig', Pipeline([extra_trees])))
pipelines.append(('Ada-orig', Pipeline([adaboost])))
pipelines.append(('GB-orig', Pipeline([gradient_boosting])))
pipelines.append(('Vot-orig', Pipeline([voting])))

# Dataset padronizado
pipelines.append(('LR-padr', Pipeline([standard_scaler, reg_log])))
```

```
pipelines.append(('KNN-padr', Pipeline([standard_scaler, knn])))
pipelines.append(('CART-padr', Pipeline([standard_scaler, cart])))
pipelines.append(('NB-padr', Pipeline([standard_scaler, naive_bayes])))
pipelines.append(('SVM-padr', Pipeline([standard_scaler, svm])))
pipelines.append(('Bag-padr', Pipeline([standard_scaler, bagging])))
pipelines.append(('RF-padr', Pipeline([standard_scaler, random_forest])))
pipelines.append(('ET-padr', Pipeline([standard_scaler, extra_trees])))
pipelines.append(('Ada-padr', Pipeline([standard_scaler, adaboost])))
pipelines.append(('GB-padr', Pipeline([standard_scaler,
gradient_boosting])))
pipelines.append(('Vot-padr', Pipeline([standard_scaler, voting])))

# Dataset normalizado
pipelines.append(('LR-norm', Pipeline([min_max_scaler, reg_log])))
pipelines.append(('KNN-norm', Pipeline([min_max_scaler, knn])))
pipelines.append(('CART-norm', Pipeline([min_max_scaler, cart])))
pipelines.append(('NB-norm', Pipeline([min_max_scaler, naive_bayes])))
pipelines.append(('SVM-norm', Pipeline([min_max_scaler, svm])))
pipelines.append(('Bag-norm', Pipeline([min_max_scaler, bagging])))
pipelines.append(('RF-norm', Pipeline([min_max_scaler, random_forest])))
pipelines.append(('ET-norm', Pipeline([min_max_scaler, extra_trees])))
pipelines.append(('Ada-norm', Pipeline([min_max_scaler, adaboost])))
pipelines.append(('GB-norm', Pipeline([min_max_scaler,
gradient_boosting])))
pipelines.append(('Vot-norm', Pipeline([min_max_scaler, voting])))

# Executando os pipelines
for name, model in pipelines:
    cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %.3f (%.3f)" % (name, cv_results.mean(), cv_results.std())
# formatando para 3 casas decimais
    print(msg)

# Boxplot de comparação dos modelos
fig = plt.figure(figsize=(25,6))
fig.suptitle('Comparação dos Modelos - Dataset original, padronizado e
```

```
normalizado')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names, rotation=90)
plt.show()
```

Saída:

```
LR-orig: 0.762 (0.057)
KNN-orig: 0.712 (0.059)
CART-orig: 0.702 (0.063)
NB-orig: 0.746 (0.051)
SVM-orig: 0.749 (0.041)
Bag-orig: 0.761 (0.047)
RF-orig: 0.772 (0.048)
ET-orig: 0.756 (0.066)
Ada-orig: 0.749 (0.053)
GB-orig: 0.751 (0.057)
Vot-orig: 0.763 (0.061)
LR-padr: 0.762 (0.057)
KNN-padr: 0.727 (0.069)
CART-padr: 0.699 (0.048)
NB-padr: 0.746 (0.051)
SVM-padr: 0.763 (0.053)
Bag-padr: 0.772 (0.045)
RF-padr: 0.751 (0.048)
ET-padr: 0.776 (0.061)
Ada-padr: 0.749 (0.053)
GB-padr: 0.748 (0.058)
Vot-padr: 0.774 (0.047)
LR-norm: 0.751 (0.045)
KNN-norm: 0.725 (0.083)
CART-norm: 0.697 (0.069)
NB-norm: 0.746 (0.051)
SVM-norm: 0.772 (0.054)
Bag-norm: 0.766 (0.050)
RF-norm: 0.776 (0.043)
ET-norm: 0.767 (0.059)
Ada-norm: 0.749 (0.053)
```

```
GB-norm: 0.749 (0.058)
Vot-norm: 0.774 (0.049)
```

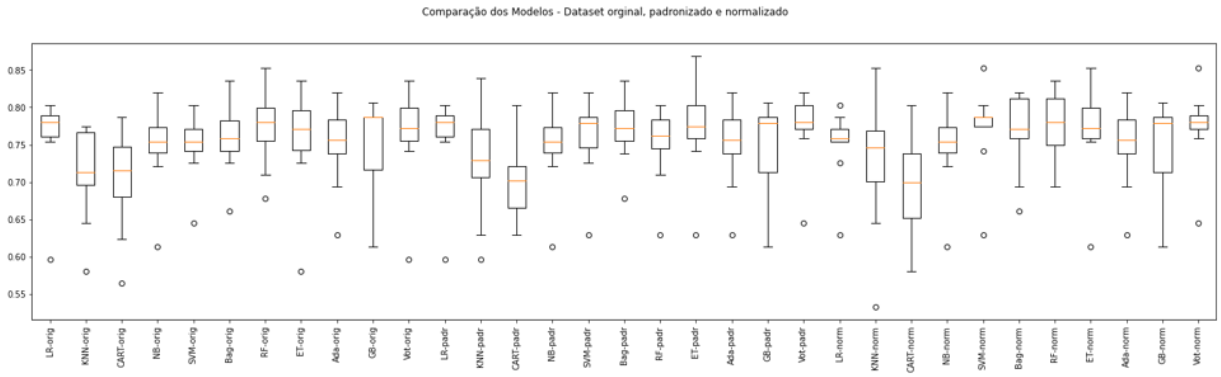


Figura 11.7: Saída.

Agora vamos utilizar a biblioteca *Grid Search* para executar a otimização dos hiperparâmetros do algoritmo KNN, a título de exemplificação, a fim de buscar resultados ainda melhores. Para isso, tentaremos todos os valores ímpares de *k* entre 1 e 21 e diferentes métricas de distância. Você pode experimentar também fazer a otimização de outros algoritmos para verificar se é possível encontrar uma configuração de modelo que supere os melhores resultados até o momento.

```
# Tuning do KNN

np.random.seed(7) # definindo uma semente global para este bloco

pipelines = []

# Definindo os componentes do pipeline
knn = ('KNN', KNeighborsClassifier())
standard_scaler = ('StandardScaler', StandardScaler())
min_max_scaler = ('MinMaxScaler', MinMaxScaler())

pipelines.append(('knn-orig', Pipeline(steps=[knn])))
pipelines.append(('knn-padr', Pipeline(steps=[standard_scaler, knn])))
pipelines.append(('knn-norm', Pipeline(steps=[min_max_scaler, knn])))
```

```
param_grid = {
    'KNN__n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
    'KNN__metric': ["euclidean", "manhattan", "minkowski"],
}

# Prepara e executa o GridSearchCV
for name, model in pipelines:
    grid = GridSearchCV(estimator=model, param_grid=param_grid,
scoring=scoring, cv=kfold)
    grid.fit(X_train, y_train)
    # imprime a melhor configuração
    print("Sem tratamento de missings: %s - Melhor: %f usando %s" % (name,
grid.best_score_, grid.best_params_))
```

Saída:

```
Sem tratamento de missings: knn-orig - Melhor: 0.757615 usando
{'KNN__metric': 'manhattan', 'KNN__n_neighbors': 17}
Sem tratamento de missings: knn-padr - Melhor: 0.757509 usando
{'KNN__metric': 'manhattan', 'KNN__n_neighbors': 19}
Sem tratamento de missings: knn-norm - Melhor: 0.759228 usando
{'KNN__metric': 'euclidean', 'KNN__n_neighbors': 13}
```

Podemos agora finalizar o modelo escolhido. Apenas para exemplificar, suponha que o modelo que alcançou os melhores resultados e foi o escolhido para implementação foi a regressão logística com os dados padronizados. A seguir, finalizaremos esse modelo, treinando-o em todo o conjunto de dados de treinamento (sem validação cruzada) e faremos predições para o conjunto de dados de teste que foi separado logo no início da prática, a fim de confirmarmos os resultados. Primeiro, vamos realizar a padronização dos dados de entrada. Depois, treinaremos o modelo e exibiremos a acurácia de teste.

```
# Avaliação do modelo com o conjunto de testes

# Preparação do modelo
scaler = StandardScaler().fit(X_train) # ajuste do scaler com o conjunto
de treino
```

```
rescaledX = scaler.transform(X_train) # aplicação da padronização no
conjunto de treino
model = LogisticRegression(max_iter=200)
model.fit(rescaledX, y_train)

# Estimativa da acurácia no conjunto de teste
rescaledTestX = scaler.transform(X_test) # aplicação da padronização no
conjunto de teste
predictions = model.predict(rescaledTestX)
print(accuracy_score(y_test, predictions))
```

Saída:

```
0.7727272727272727
```

Por meio do conjunto de teste, verificamos que alcançamos uma acurácia de 77,27% em dados não vistos. Esse resultado foi ainda melhor do que a nossa avaliação anterior da regressão logística, que alcançou acurácia média de 76,2%. Valores semelhantes à acurácia de teste são esperados quando esse modelo estiver executando em produção e fazendo predições para novos dados.

Vamos agora preparar o modelo para utilização em produção. Para isso, vamos treiná-lo com todo o *dataset*, e não apenas o conjunto de treino.

```
# Preparação do modelo com TODO o dataset
scaler = StandardScaler().fit(X) # ajuste do scaler com TODO o dataset
rescaledX = scaler.transform(X) # aplicação da padronização com TODO o
dataset
model.fit(rescaledX, y)
```

Finalmente, vamos simular a aplicação do modelo em dados não vistos, imaginando que chegaram 3 novas instâncias, mas que não sabemos a classe de saída. Poderemos então aplicar nosso modelo recém-treinado para fazer a predição das classes. Para isso, será necessário antes padronizar os dados usando a mesma escala dos dados usados no treinamento do modelo:

```
# Novos dados - não sabemos a classe!
data = {'preg':  [1, 9, 5],
        'plas': [90, 100, 110],
        'pres': [50, 60, 50],
        'skin': [30, 30, 30],
        'test': [100, 100, 100],
        'mass': [20.0, 30.0, 40.0],
        'pedi': [1.0, 2.0, 1.0],
        'age': [15, 40, 40],
        }

atributos = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi',
'age']
entrada = pd.DataFrame(data, columns=atributos)

array_entrada = entrada.values
X_entrada = array_entrada[:,0:8].astype(float)

# Padronização nos dados de entrada usando o scaler utilizado em X
rescaledEntradaX = scaler.transform(X_entrada)
print(rescaledEntradaX)
```

Saída:

```
[[-0.84488505 -0.96691063 -0.98770975  0.59362962  0.17539902 -1.52208897
   1.59499624 -1.55207596]
 [ 1.53084665 -0.65393918 -0.47073225  0.59362962  0.17539902 -0.25289651
   4.61511492  0.57511787]
 [ 0.3429808  -0.34096773 -0.98770975  0.59362962  0.17539902  1.01629594
   1.59499624  0.57511787]]
```

Podemos agora realizar a predição das classes para os novos dados:

```
# Predição de classes dos dados de entrada
saidas = model.predict(rescaledEntradaX)
print(saidas)
```

Saída:

```
[0. 1. 1.]
```

Exemplo prático de *feature selection* em Python

Nesta prática, vamos aplicar o processo de seleção de atributos, também conhecido como *feature selection*. Para isso, vamos trabalhar novamente com o *dataset Diabetes*. Todo o código está comentado, para facilitar o entendimento.

Iniciaremos esta prática importando os pacotes necessários:

```
# Imports
import pandas as pd
import numpy as np
from numpy import set_printoptions
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import ExtraTreesClassifier

# Novos imports
from sklearn.feature_selection import SelectKBest # para a seleção
univariada
from sklearn.feature_selection import f_classif # para o teste ANOVA da
seleção univariada
from sklearn.feature_selection import RFE # para a eliminação recursiva de
atributos
```

Vamos então carregar o *dataset*, converter seus dados para um *dataframe*, adicionar a coluna *target* e exibir as primeiras linhas para checarmos se tudo foi carregado com sucesso.

```
# Carrega arquivo csv usando Pandas por meio de uma URL

# Informa a URL de importação do dataset
url =
"https://raw.githubusercontent.com/profkalinowski/livroescd/main/diabetes.
csv"

# Informa o cabeçalho das colunas
```

```
colunas = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']

# Lê o arquivo utilizando as colunas informadas
dataset = pd.read_csv(url, names=colunas, skiprows=1, delimiter=',')
```

Em seguida, vamos preparar nossos dados para a aplicação dos métodos de seleção de atributos.

```
# separando os atributos e a classe do dataset
array = dataset.values
X = array[:,0:8]
y = array[:,8]

# A semente (seed) pode ser qualquer número, e garante que os resultados
possam ser reproduzidos de forma idêntica toda vez que o script for
rodado.
# Isso é muito importante quando trabalhamos com modelos ou métodos que se
utilizam de algum tipo de aleatoriedade.
seed = 7
```

Seleção univariada

A função `SelectKBest()` pode ser usada com diversos testes estatísticos para selecionar os atributos. Vamos usar o teste *ANOVA F-value* e selecionar os 4 melhores atributos que podem ser usados como variáveis preditoras.

Neste contexto, o teste *ANOVA F-value* estima o quanto cada característica de *X* é dependente da classe *y*, e é um teste especialmente adequado quanto temos a variável de saída categórica e as variáveis de entrada numéricas, como neste *dataset*.

```
# Função para seleção de atributos
best_var = SelectKBest(score_func=f_classif, k=4)

# Executa a função de pontuação em (X, y) e obtém os atributos
selecionados
fit = best_var.fit(X, y)
```

```
# Reduz X para os atributos selecionados
features = fit.transform(X)

# Resultados
print('\nNúmero original de atributos:', X.shape[1])
print('\nNúmero reduzido de atributos:', features.shape[1])

# Exibe os atributos originais
print("\nAtributos Originais:", dataset.columns[0:8])

# Exibe as pontuações de cada atributo e os 4 escolhidos (com as
pontuações mais altas): preg, plas, mass e age
set_printoptions(precision=3) # 3 casas decimais
print("\nScores dos Atributos Originais:", fit.scores_)
print("\nAtributos Selecionados:",
best_var.get_feature_names_out(input_features=dataset.columns[0:8]))

# Imprime o dataset apenas com as colunas selecionadas
print("\n", features)
```

Saída:

```
Número original de atributos: 8

Número reduzido de atributos: 4

Atributos Originais: Index(['preg', 'plas', 'pres', 'skin', 'test',
'mass', 'pedi', 'age'], dtype='object')

Scores dos Atributos Originais: [ 39.67  213.162   3.257   4.304  13.281
71.772  23.871  46.141]

Atributos Selecionados: ['preg' 'plas' 'mass' 'age']

 [[  6.  148.   33.6  50. ]
 [  1.   85.   26.6  31. ]
 [  8.  183.   23.3  32. ]
 ...
 [  5.  121.   26.2  30. ]
```

```
 [  1.  126.   30.1  47. ]
 [  1.   93.   30.4  23. ]]
```

Para saber mais sobre esse teste, consulte: https://www.statisticshowto.com/probability-and-statistics/f-statistic-value-test/.

Eliminação recursiva de atributos

Vamos aplicar a técnica de eliminação recursiva de atributos com um algoritmo de regressão logística (poderia ser qualquer classificador) para selecionar as 4 melhores variáveis preditoras.

```
# Criação do modelo
modelo = LogisticRegression(max_iter=200)

# Eliminação recursiva de variáveis
rfe = RFE(modelo, n_features_to_select=4)
fit = rfe.fit(X, y)

# Print dos resultados
print("Atributos Originais:", dataset.columns[0:8])

# Exibe os atributos selecionados (marcados como True em "Atributos
Selecionados"
# e com valor 1 em "Ranking dos Atributos"): preg, plas, mass e pedi.
# (Basta mapear manualmente o índice dos nomes dos respectivos atributos)
print("\nAtributos Selecionados: %s" % fit.support_)
print("\nRanking de atributos: %s" % fit.ranking_)
print("\nQtd de melhores Atributos: %d" % fit.n_features_)
print("\nNomes dos Atributos Selecionados: %s" %
fit.get_feature_names_out(input_features=dataset.columns[0:8]))
```

Saída:

```
Atributos Originais: Index(['preg', 'plas', 'pres', 'skin', 'test',
'mass', 'pedi', 'age'], dtype='object')

Atributos Selecionados: [ True  True False False False  True  True False]
```

```
Ranking de atributos: [1 1 3 5 4 1 1 2]

Qtd de melhores Atributos: 4

Nomes dos Atributos Selecionados: ['preg' 'plas' 'mass' 'pedi']
```

Importância de atributos com *ExtraTrees*

Vamos construir um classificador *ExtraTreesClassifier* para calcular a importância dos atributos e, posteriormente, selecionar quais vamos utilizar.

```
# Criação do modelo para seleção de atributos
modelo = ExtraTreesClassifier(n_estimators=100)
modelo.fit(X,y)

# Exibe os atributos originais
print("\nAtributos Originais:", dataset.columns[0:8])

# Exibe a pontuação de importância para cada atributo (quanto maior a
pontuação, mais importante é o atributo).
# Atributos selecionados: plas, mass, pedi, age
print(modelo.feature_importances_)
```

Saída:

```
Atributos Originais: Index(['preg', 'plas', 'pres', 'skin', 'test',
'mass', 'pedi', 'age'], dtype='object')
[0.109 0.231 0.1   0.081 0.073 0.14  0.122 0.144]
```

Resumo dos atributos selecionados:

- Técnica 1: preg, plas, mass e age
- Técnica 2: preg, plas, mass e pedi
- Técnica 3: plas, mass, pedi, age

Observação: É possível utilizar técnicas de *feature selection* como parte de um *pipeline*. Veja um exemplo no trecho de código a seguir:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
  ('feature_selection', SelectKBest(score_func=f_classif, k=4)),
  ('classification', RandomForestClassifier(n_estimators=100,
max_features=3, random_state=7))
])
```

Usando técnicas de *feature selection*, podemos decidir quais atributos queremos selecionar para que sejam utilizados no treinamento do modelo de *Machine Learning*, o que seria o passo seguinte. Nesta prática, vimos como utilizar três diferentes técnicas de *feature selection*. Note que, dependendo da técnica utilizada, os atributos selecionados poderão ser diferentes.

Parte V – Arquitetura, Projeto e Controle da Qualidade

Capítulo 12
Implantação de modelos de Machine Learning

Uma das atividades finais no processo de desenvolvimento de sistemas de software inteligentes é implantar o modelo e decidir como ele será usado em produção. A implantação envolve disponibilizar modelos treinados, avaliados e aprovados. Neste capítulo, entenderemos como os modelos de *Machine Learning* (ML) são implantados na prática e quais estratégias podem ser usadas para isso.

12.1 Formas de implantação de modelos de Machine Learning

Há três formas principais de implantação de modelos de *Machine Learning*: implantar o modelo de forma embarcada, implantar o modelo como um serviço separado e implantar o modelo utilizando uma solução *Platform-as-a-Service* (PaaS). Veremos mais sobre essas formas a seguir.

O **modelo embarcado** envolve instalar e executar o modelo no hardware e na infraestrutura da própria organização. Isso pode fornecer mais controle sobre o modelo e os dados que ele usa, mas também exige que a organização invista e mantenha o hardware e a infraestrutura necessários. Nessa forma de implantação, o modelo é empacotado na aplicação que o consome.

Por exemplo, em aplicações web, o modelo poderia ser carregado pelo código do back-end de uma aplicação *on-premise*, que também invocará as previsões do modelo quando solicitado pelo usuário (por exemplo, em um front-end). No caso de uma aplicação web construída em Python, o back-end poderia ser construído utilizando frameworks como *Django* ou *Flask*. Nessa categoria temos ainda os *data apps* (por exemplo, *Streamlit*), que visam facilitar a criação de aplicações para visualizações simplificadas.

O **modelo implantado como um serviço separado** envolve hospedar o modelo na infraestrutura de um provedor de nuvem, como *Amazon Web Services* (AWS), *Microsoft Azure* ou *Google Cloud Platform*. Isso pode ser mais econômico e escalável do que uma implantação local, mas requer uma conexão confiável com a internet e pode envolver a dependência do provedor.

Em exemplos dessa forma de implantação, o modelo, que é um serviço independente das aplicações que o consomem, pode ser hospedado na nuvem (*serverless*) ou em um servidor da organização (*on-premise*). O modelo é então acionado como um componente à parte pela aplicação que necessita de suas predições.

Platform-as-a-Service (PaaS) envolve utilizar soluções de fornecedores de serviços em nuvem específicas para *Machine Learning*. Entre as opções mais conhecidas estão o *Amazon SageMaker*, o *Google Cloud Platform* e o *Azure Machine Learning*. Essas soluções proveem opções para disponibilizar os modelos diretamente para o usuário final, embora os modelos possam também ser disponibilizados para que sejam consumidos por outras aplicações.

Um exemplo de implantação desse tipo envolve empacotar o modelo e suas dependências em um contêiner Docker fornecido pelo provedor, que pode ser implantado em qualquer infraestrutura compatível. Essa pode ser uma maneira flexível e portátil de implantar modelos de *Machine Learning*, mas requer alguma familiaridade com as tecnologias de contêiner.

Dependendo do uso pretendido dos modelos, pode haver outros tipos de implantação, como a de borda (*edge*), que envolve a implantação do modelo em dispositivos na borda de uma rede, como dispositivos IoT ou servidores de borda. Isso pode ser útil para aplicativos que exigem baixa latência ou precisam operar off-line, mas também pode ser mais difícil de configurar e gerenciar.

A figura a seguir ilustra as formas de implantação. É bastante comum que a análise exploratória, o pré-processamento, o treinamento e a avaliação ocorram utilizando programação literária interativa, os famosos *notebooks*. Uma vez que se chega a um modelo adequado, ele é então implantado utilizando uma das opções de implantação, ficando disponível para ser consumido por aplicações finais que podem ser executadas em diferentes tipos de dispositivos.

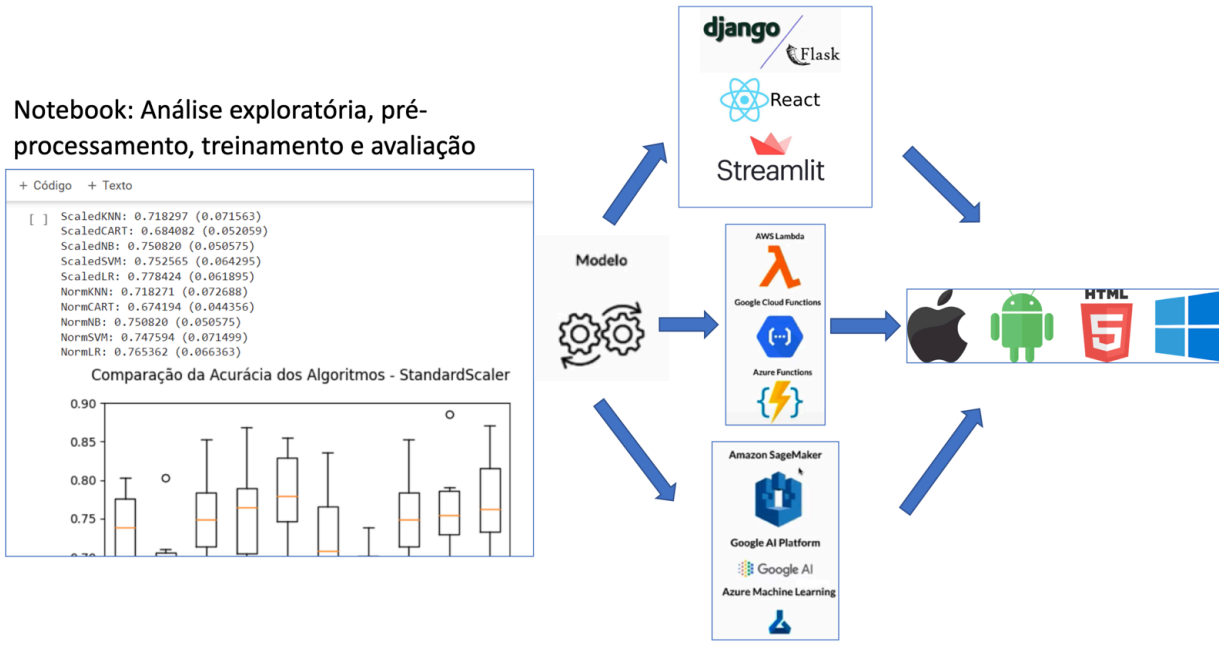


Figura 12.1: Principais formas de implantação de modelos de Machine Learning.

Agora que sabemos as formas mais comuns para implantar modelos de *Machine Learning*, é útil conhecer as vantagens e desvantagens das estratégias que envolvem o uso da infraestrutura de um provedor, pois, dependendo do contexto da solução e do conjunto

de requisitos da equipe de Ciência de Dados, uma implantação na nuvem pode ser mais benéfica do que uma implantação local com um modelo embarcado.

Vantagens da implantação em nuvem:

- **Acesso a recursos:** os provedores de nuvem oferecem uma variedade de recursos de hardware e software que podem ser usados para treinar e implantar modelos de *Machine Learning*, incluindo GPUs, TPUs e uma variedade de estruturas e bibliotecas pré-instaladas;
- **Escalabilidade:** as plataformas de nuvem podem facilmente aumentar ou diminuir para atender às necessidades de suas cargas de trabalho de *Machine Learning*, permitindo que você lide com picos repentinos de demanda sem precisar investir em novo hardware;
- **Eficiência de custos:** os provedores de nuvem oferecem uma variedade de opções de preços, incluindo preços pré-pagos e baseados em assinatura, que podem ajudar você a minimizar custos e otimizar o orçamento das organizações;
- **Colaboração e compartilhamento:** as plataformas em nuvem facilitam a colaboração e o compartilhamento de modelos de *Machine Learning* e dados pelas equipes, independentemente de sua localização;
- **Segurança:** os provedores de nuvem têm medidas de segurança robustas para proteger seus dados e modelos de *Machine Learning*, incluindo criptografia de dados, controles de acesso e auditorias de segurança regulares.

Desvantagens da implantação em nuvem:

- **Dependência da internet:** para usar os serviços de *Machine Learning* baseados em nuvem, você deve ter uma conexão estável com a internet. Isso pode ser uma desvantagem em áreas com acesso limitado ou não confiável à internet;
- **Potencial para bloqueio de fornecedor:** dependendo do provedor de nuvem, você pode se tornar dependente de sua

plataforma e serviços, o que pode dificultar a mudança para um provedor diferente no futuro;
- **Latência:** dependendo da localização dos servidores em nuvem e da distância de seus usuários, pode haver um atraso no tempo de resposta de seus modelos de *Machine Learning*, o que pode impactar no desempenho deles;
- **Custo:** embora as plataformas de nuvem possam ser econômicas em alguns casos, elas também podem ser caras se tiver um grande volume de dados ou um grande número de implantações de modelo de *Machine Learning*. Neste caso, é importante otimizar o uso de recursos nas plataformas de *Machine Learning*;
- **Preocupações com segurança:** embora os provedores de nuvem tenham fortes medidas de segurança, sempre há o risco de violações de dados ou ataques cibernéticos;
- **Complexidade:** configurar e gerenciar um modelo de *Machine Learning* na nuvem pode ser mais complexo do que no local, especialmente para organizações que não estão familiarizadas com as tecnologias de nuvem.

12.2 Armazenamento de modelos de Machine Learning em arquivos

Em *Machine Learning*, é bastante comum que se faça necessário treinar e salvar nossos modelos em um arquivo e restaurá-los para, por exemplo, testá-los com novos dados, ou comparar o desempenho de vários modelos. Esse procedimento de salvar o modelo é conhecido como **serialização de objetos**, que é basicamente uma representação de um objeto com um fluxo de *bytes*, a fim de armazená-lo em disco, enviá-lo por uma rede ou salvá-lo em um banco de dados. Por outro lado, o procedimento de restauração/carregamento do modelo é conhecido como **desserialização**.

Nesta seção, exploraremos duas maneiras de salvar e carregar modelos de *Machine Learning* em Python: a abordagem *Pickle* e a biblioteca *Joblib*.

Pickle

Pickle é uma biblioteca embutida em Python que, como já falamos, permite a serialização e desserialização de objetos. O nome vem do verbo em inglês "*to pickle*", que significa preservar algo para uso posterior. Da mesma forma, os objetos Python são serializados em arquivos para uso posterior.

O trecho de código a seguir armazena um modelo treinado que se encontra na variável `modelo` em um arquivo chamado `pickle_classificador.pkl` utilizando a biblioteca `pickle`. Note que basicamente foi aberto um arquivo para escrita em formato binário `('wb')` e que o método `dump` da biblioteca `pickle` se encarrega de passar o conteúdo do modelo de forma serializada para o arquivo que foi aberto. Depois, basta fechar o arquivo.

```
# Salvando o modelo treinado em um arquivo usando a biblioteca Pickle
import pickle
pickle_out = open('pickle_classificador.pkl', 'wb')
pickle.dump(modelo, pickle_out)
pickle_out.close()
```

Após a execução desse código em uma célula de um *notebook*, o arquivo `pickle_classificador.pkl` existirá na sua máquina, contendo a serialização do seu modelo. A partir daí, para carregar o modelo em um objeto Python que possa realizar uma predição utilizando esse modelo, basta disponibilizar o arquivo seguindo as linhas abaixo.

Desta vez, o código envolve a abertura de um arquivo para leitura em formato binário `('rb')` e o método `load` da biblioteca `pickle` se encarrega de recuperar o conteúdo do arquivo serializado e de passá-lo para a variável `modelo`. Com isso, a predição pode ser feita acionando o método `predict` do modelo. Neste exemplo, foi carregado um modelo que faz a classificação de um cliente em um

bom ou mau pagador, com base na renda mensal, idade, número de dependentes e valor consignado em empréstimos vigentes.

```
# Usando a inferência de um modelo usando a biblioteca Pickle
import pickle
# Carregando o modelo treinado
pickle_in = open('pickle_classificador.pkl', 'rb')
modelo = pickle.load(pickle_in)
pickle_in.close()

# Utilizando o modelo para fazer uma predição
Predicao = modelo.predict([[rendaMensal, idade, numeroDependentes,
valorConsignado]])
```

Joblib

Joblib é um conjunto de ferramentas para criar *pipelines* em Python. A biblioteca é otimizada para ser rápida e robusta em grandes volumes de dados e possui otimizações específicas para *arrays NumPy*. Nos scripts Python a seguir, mostraremos como salvar e carregar modelos de ML usando *Joblib*. O primeiro passo que precisa ser feito é importar a biblioteca *Joblib*, do mesmo jeito que foi importada a biblioteca *Pickle*. Feito isso, podemos salvar o conteúdo da variável `modelo` em um arquivo chamado `joblib_classificador.pkl` .

```
# Salvando o modelo treinado em um arquivo usando a biblioteca Joblib
import joblib
joblib_file = 'joblib_classificador.pkl'
joblib.dump(modelo, joblib_file)
```

O resultado da execução desse código é um arquivo chamado `joblib_classificador.pkl` que foi gerado no diretório de trabalho atual. Agora, para carregar o modelo salvo usando *Joblib* e consumi-lo (gerar previsões), basta usar o método `load` da biblioteca *Joblib*, que se encarrega de desserializar o arquivo e de passá-lo para a variável `modelo` , e, na sequência, usar o método `predict` . Veja que o procedimento é muito parecido com a abordagem *Pickle*.

```
# Usando a inferência de um modelo usando a biblioteca Joblib
import joblib

# Carregando o modelo treinado
modelo = joblib.load('joblib_classificador.pkl')

# Utilizando o modelo para fazer uma predição
predicao = modelo.predict([[rendaMensal, idade, numeroDependentes,
valorConsignado]])
```

Resumindo, a implementação para salvar e carregar modelos de ML em arquivos é simples. Para decidir qual biblioteca usar, considere o seguinte: *Joblib* é mais rápido ao salvar/carregar grandes matrizes *NumPy*, enquanto *Pickle* é mais rápido com grandes coleções de objetos Python. Portanto, se o seu modelo contiver grandes matrizes *NumPy* (como a maioria dos modelos), o *Joblib* deve ser mais rápido. Para instâncias relativamente pequenas, não faz diferença escolher uma abordagem sobre a outra.

12.3 Implantação de modelos de Machine Learning embarcados

Agora que sabemos como disponibilizar modelos de ML em arquivos, podemos nos aprofundar mais nas diferentes formas de implantação. Nesta seção, veremos como funciona a transição de um ambiente de experimentação para um ambiente de produção implantando um modelo embarcado no back-end de uma aplicação web. A figura a seguir apresenta os componentes envolvidos desde um modelo treinado, que é criado no ambiente de experimentação, até um modelo implantado, que é integrado com o resto das funcionalidades de um sistema como um todo.

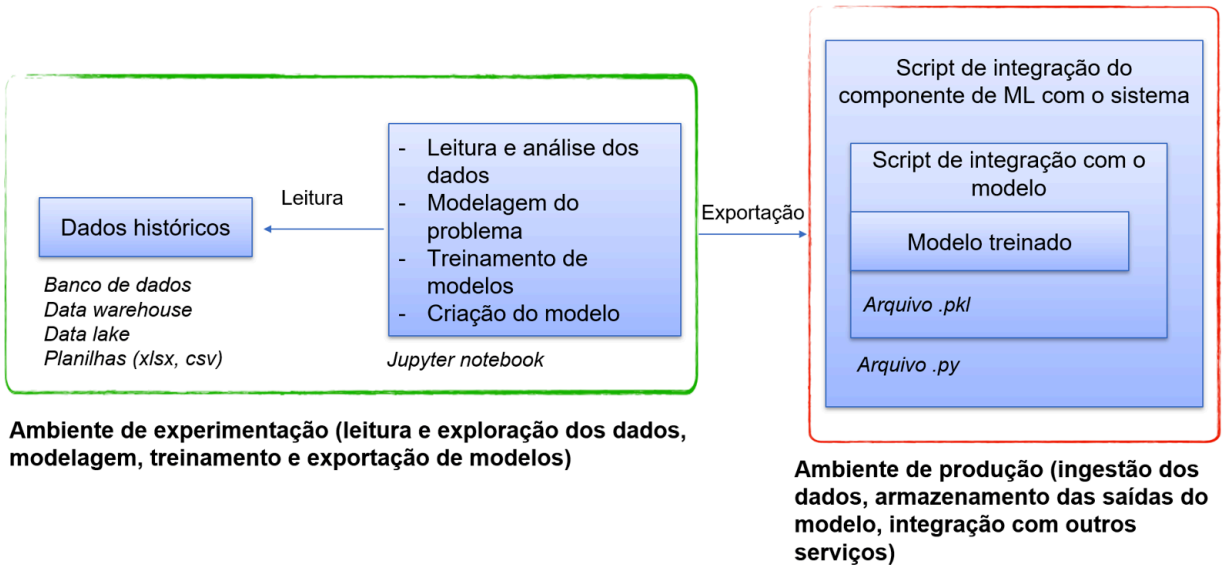


Figura 12.2: Implantação de um modelo de ML embarcado.

Veja que é no ambiente de produção onde o sistema de software inteligente orquestra a interação dos diferentes componentes, por exemplo: a ingestão dos dados que o modelo precisa; a invocação do modelo para produzir previsões; o armazenamento das previsões; o monitoramento dos resultados do modelo e a integração com outros serviços (modelos, heurísticas), caso seja necessário.

Como mostrado na figura, para implantar um modelo de ML embarcado, é necessário configurar um ambiente de experimentação para:

1. Conectar com a fonte de dados, fazer a leitura, exploração e visualização dos dados. Tipicamente, *notebooks* são usados para executar *scripts* para esse fim;
2. Modelar o problema, treinar e testar diferentes algoritmos de ML com os dados analisados e processados. Aqui, tipicamente é usado *Scikit-learn*, uma biblioteca Python de código aberto que inclui vários algoritmos de classificação, regressão e agrupamento, além de interagir com outras bibliotecas numéricas e científicas como *NumPy* e *SciPy*;
3. Exportar o modelo escolhido (melhor desempenho de acordo com os objetivos do modelo e as expectativas do cliente) como

um arquivo. Bibliotecas como *Pickle* e *Joblib*, detalhadas neste capitulo, são usadas para isso.

Já no ambiente de produção precisaremos:

1. Criar uma aplicação web ou um *script* independente que integra diferentes componentes envolvidos no ambiente de experimentação. Por exemplo, a captura dos dados de operação (equivalente aos dados de treinamento), o carregamento do modelo escolhido (leitura do arquivo *.pkl* ou *.joblib*) e o fornecimento das previsões. Isso é chamado de *script* de integração com o modelo, que, no final das contas, é um componente de ML funcional;
2. Integrar o componente de ML com o sistema ou outros componentes que podem fazer uso das previsões do modelo. Por exemplo: outros modelos, aplicação de heurísticas e visualização das saídas do modelo;
3. Testar a aplicação web, o *script* e as diferentes integrações localmente para garantir que funcione conforme o esperado.

O trecho do código a seguir apresenta um exemplo de implantação de um modelo embarcado, mais especificamente, um *script Python* que integra um modelo treinado no back-end de uma aplicação web *Flask*, que disponibiliza uma `API predict` para consumir o modelo com dados que vêm em uma requisição web, e que no final retorna as previsões do modelo.

```
# Importando bibliotecas necessárias
from sklearn.linear_model import LogisticRegression
import joblib
from flask import Flask, request

# Treinando o modelo
X_train = ... # Características de entrada para treinamento
Y_train = ... # Rótulos-alvo para treinamento
modelo = LogisticRegression()
modelo.fit(X_train, y_train)
```

```
# Salvando o modelo treinado em um arquivo
joblib.dump(model, 'model.pkl')

# Criando uma aplicação Flask
App = Flask(__name__)

# Criando um ponto final de API para fazer previsões
@app.route('/predict', methods=['POST'])
def predict():
    # Carregando o modelo treinado a partir de um arquivo
    modelo = joblib.load('model.pkl')

    # Extraindo as características de entrada da solicitação
    X_input = request.json['input']

    # Usando o modelo carregado para fornecer as previsões
    y_pred = modelo.predict(X_input)

    # Retornando as previsões
    return y_pred
```

12.4 Implantação de modelos de Machine Learning como serviço separado na nuvem

Implantar modelos de ML como um serviço separado na nuvem tem se tornado uma prática comum na indústria. O principal benefício desse tipo de implantação é que ele permite aos usuários ter um maior acesso ao modelo, além da capacidade de dimensionar o modelo para lidar com um grande número de solicitações. Isso pode ser especialmente útil para sistemas de software inteligentes que exigem previsões em tempo real ou são usados por um grande número de usuários.

No entanto, uma possível desvantagem da implantação de modelos de ML na nuvem é o *custo*. Dependendo do provedor de nuvem e da quantidade de recursos exigidos pelo modelo, usar a nuvem pode ser mais caro do que hospedar o modelo *on-premise*. Além disso, pode haver preocupações com a privacidade e segurança dos dados ao usar a nuvem, pois dados confidenciais podem ser armazenados e processados em servidores remotos.

A figura a seguir apresenta os componentes envolvidos na transição de um modelo treinado a um modelo implantado na nuvem. Veja que o processo de criação do modelo não muda, pois é baseado em um ambiente de experimentação que envolve atividades típicas de projetos de Ciência de Dados como análise exploratória, modelagem e treinamento de modelos.

Como já vimos, essas atividades são desenvolvidas em *notebooks* que podem ser hospedados nas plataformas de ML de fornecedores como *Microsoft*, *Google* ou *Amazon*. Neste exemplo, o modelo treinado é disponibilizado como um *endpoint* onde são recebidas chamadas de API para consumir o modelo e fornecer as respostas. No ambiente de produção, componentes na nuvem podem ser utilizados para suportar a transferência, processamento, armazenamento e monitoramento dos dados.

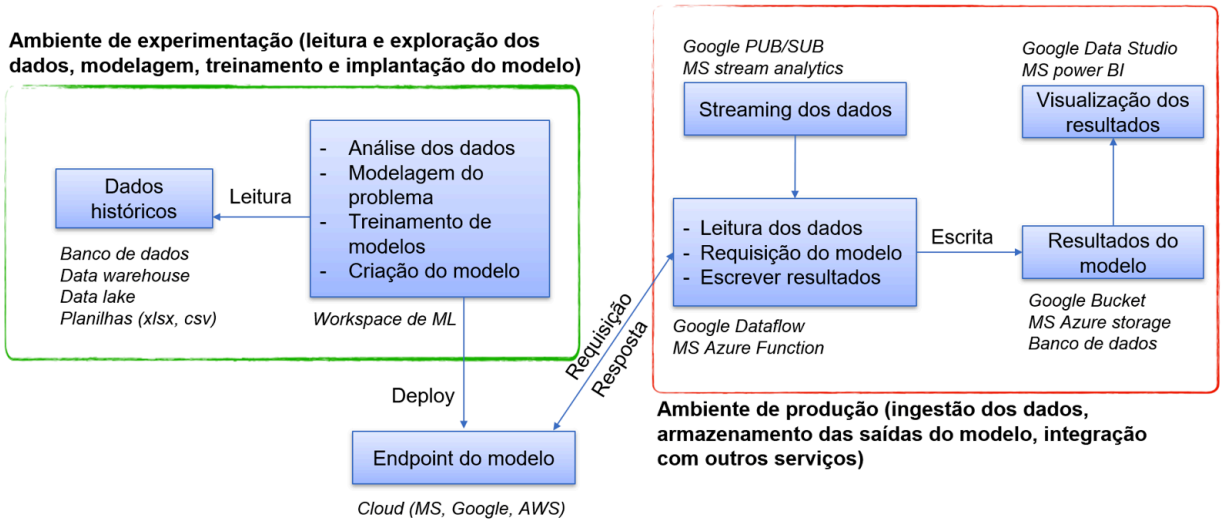


Figura 12.3: Implantação de um modelo de Machine Learning na nuvem.

Lembre-se de que, para usar os serviços de computação na nuvem, é necessário ter uma conta ativa no site do provedor de infraestrutura como *Google* ou *Microsoft*. A maioria dos serviços de armazenamento e processamento são pagos. Porém, em alguns casos, os usuários novos dessas plataformas têm direito a um número de créditos que pode ser usado em um tempo determinado por cada provedor. Para implantar um modelo, geralmente é necessário fazer uso da interface gráfica da plataforma de ML do provedor ou do SDK para:

- Criar ou importar um modelo de ML na plataforma;
- Registrar o modelo de ML na plataforma;
- Definir as configurações dos recursos de computação que serão usados durante a implantação e operação do modelo de ML;
- Criar um *endpoint* para hospedar o modelo de ML;
- Implantar o modelo de ML no *endpoint* criado.

O trecho do código a seguir apresenta um *script Python* com os passos descritos anteriormente para implantar um modelo na plataforma de ML na nuvem do *Google*.

```
# Importando as bibliotecas para conectar com a Google Cloud
# Lembre-se de instalar o módulo → pip install google-cloud-aiplatform
```

```
import pickle
from google.cloud import aiplatform
from google.cloud import storage
from google.oauth2 import service_account

# Conectando com a plataforma de computação do Google
# O arquivo json é gerado na plataforma do Google e contém os dados de
conexão com a nuvem
gc_credentials=service_account.Credentials.from_service_account_file('goog
le_cloud_secrets.json')

# Usando o módulo 'storage' para conectar com o serviço de armazenamento
do Google
# Id do projeto que estará relacionado com o modelo que será implantado
project_id = 'project-01'
storage_client=storage.Client(project=project_id,
                              credentials=gc_credentials)

# Usando o método 'bucket' do objeto 'storage_client' para armazenar o
modelo de ML que foi exportado usando a biblioteca Pickle
# Especificando o nome do bucket e do arquivo que será armazenado
bucket_name = 'bucket-test'
model_name = 'model.pkl'

# Conectando com o bucket especificado
bucket = storage_client.bucket(bucket_name)

# Especificando o caminho onde será salvo o arquivo dentro do bucket
destination_blob_name = 'model/{}'.format(bucket_name)

# Fazendo o upload do arquivo no bucket
blob = bucket.blob(destination_blob_name)
blob.upload_from_filename(model_name)

print(f"Arquivo {model_name} carregado no caminho
{destination_blob_name}")

# Inicializando o workspace de inteligência artificial da Google Cloud
para registrar um modelo e criar um endpoint
```

```
# Especificando a região
region = 'us-east1'

# Init vertex ai
Aiplatform.init(project = project_id,
                location = region,
                credentials = gc_credentials,
                staging_bucket = 'gs://models-storage')

# Importando um modelo - neste caso, o modelo exportado usando a
biblioteca Pickle, no workspace de inteligência artificial do Google
Model = aiplatform.Model.upload(display_name = 'model-v1',
                                description = 'Modelo de teste',
                                artifact_uri = 'gs://models-
storage/models/',

serving_container_image_uri='docker.pkg.dev/')

# Criando um endpoint no workspace de inteligência artificial do Google
onde será hospedado o nosso modelo
endpoint=aiplatform.Endpoint.create(display_name='endpoint',
                                    project = project_id,
                                    location = region)

# Implantação do modelo no endpoint que foi criado
endpoint.deploy(model, machine_type='n1-standard-2')

# Obtendo previsões do endpoint que foi implantando na Google Cloud
endpoint.predict(X_test.tolist())
```

Ao longo deste capítulo vimos como armazenar modelos de ML em arquivos para depois carregá-los e usá-los de acordo com as nossas necessidades. Também vimos como implantar modelos embarcados e como disponibilizar modelos de ML na nuvem. Um *notebook* com a implementação desses procedimentos pode ser encontrado no link:

https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_persistencia_e_implantacao_de_modelos.ipynb.

Capítulo 13
Arquitetura de sistemas de software inteligentes

Neste capítulo discutiremos a definição e os conceitos básicos de arquitetura de software e faremos considerações sobre a arquitetura de sistemas de software inteligentes, incluindo os desafios arquiteturais reportados para sistema de *Machine Learning*.

Forneceremos ainda um exemplo de uma arquitetura de um sistema de software inteligente, inspirado em um projeto real recentemente entregue pela iniciativa ExACTa PUC-Rio. Veremos exemplos de arquiteturas que podem ser empregadas para diferentes tipos de previsão dos modelos de *Machine Learning*, incluindo predições off-line e predições em tempo real.

Por fim, discutiremos a aplicação de táticas arquiteturais que podem ser usadas para a otimização das predições dos modelos de *Machine Learning*.

13.1 Conceitos básicos de arquitetura de software

A arquitetura de software enfatiza a documentação de decisões de projeto de alto nível relacionadas com estratégias organizacionais para lidar com os requisitos não funcionais e características transversais que afetam o sistema como um todo. O projeto arquitetural pode ser entendido como o processo de identificar subsistemas que compõem um sistema e o arcabouço para controlar

esses subsistemas e sua comunicação. A saída deste processo é a descrição da arquitetura do software.

A arquitetura é também uma forma de comunicação sobre decisões técnicas estratégicas, já que muitos *stakeholders* têm interesse nessas decisões, mas não têm interesse em decisões técnicas de baixa granularidade, envolvendo o projeto detalhado ou codificação. O projeto detalhado mais próximo da codificação será abordado no capítulo seguinte.

No livro *Software Architecture in Practice*, Bass, Clemens e Kazman (2012) definem que "*a arquitetura de software de um sistema é o conjunto de estruturas requeridas para raciocinar sobre o sistema, que envolve elementos de software, relações entre eles, e as propriedades de ambos*". Embora essa definição permita o entendimento, a consideramos bastante abstrata. Gostamos particularmente da definição dada por Ian Gorton (2011) em seu livro *Essential Software Architecture*, em que ele define arquitetura de software como *a abstração que define a estrutura da solução, especifica a comunicação entre os componentes e trata os requisitos não funcionais*. Essa definição explicita a relevância dos **requisitos** e, sem dúvida, um bom arquiteto de software ou uma boa arquiteta de software deve ter os requisitos como seus melhores amigos. Um projeto arquitetural que não considera adequadamente os requisitos (principalmente os não funcionais, como destacado na definição) está fadado ao fracasso. Ou seja, a arquitetura afeta as funcionalidades do sistema de maneira geral.

Exemplos de preocupações arquiteturais são:

- Quais tecnologias (linguagens, *frameworks* etc.) devem ser utilizadas para a construção do sistema de modo que ele atenda aos requisitos não funcionais?
- Qual deve ser a estrutura geral (ou estilo arquitetural - em camadas, baseado em eventos etc.) a ser seguida para subsidiar a construção do sistema como um todo de modo que ele atenda aos requisitos não funcionais?

- Quais estratégias ou táticas de construção devem ser utilizadas para a construção do sistema de modo que ele atenda aos requisitos não funcionais?
- Quais são os principais componentes do sistema e suas respectivas interfaces?
- Como o sistema será decomposto em módulos?
- Que estratégia de comunicação deve ser usada?
- Que estilos arquiteturais são adequados?
- Como a arquitetura deve ser documentada?

Ou ainda, instanciando para requisitos não funcionais específicos:

- De que forma podemos garantir que o sistema atenda aos requisitos gerais de segurança?
- De que forma podemos garantir que o sistema atenda aos requisitos gerais de usabilidade?
- De que maneira podemos organizar a arquitetura do sistema considerando a necessidade de previsão off-line dos modelos de *Machine Learning*?

Esperamos que esses exemplos de questionamentos tenham deixado as preocupações típicas de um(a) arquiteto(a) de software mais claras. Falamos que os requisitos são os melhores amigos do arquiteto de software. Vejamos a seguir como os requisitos devem ser considerados durante o projeto arquitetural.

Requisitos funcionais

Os *requisitos funcionais* são mapeados em componentes e interfaces que refletem os serviços providos por cada componente. Durante essa definição dos componentes, um princípio a ser seguido é o ***Separation of Concerns*** (ou separação de interesses), que tem como objetivo estabelecer um sistema bem organizado, onde cada parte desempenha um papel com um significado claro e intuitivo, enquanto se maximiza a habilidade de se adaptar a mudanças.

Outra preocupação arquitetural ao pensar na distribuição das funcionalidades pelo sistema e na maneira como elas serão

oferecidas aos usuários é manter a **integridade conceitual** do projeto como um todo, que significa que o sistema não pode ser um amontoado de funcionalidades sem coerência entre elas. Um produto com integridade conceitual reflete um conjunto de ideias que são aplicadas de forma consistente sobre as suas diferentes partes. Assim, um usuário acostumado a usar uma parte do sistema se sentirá confortável ao usar outra parte do mesmo sistema.

Requisitos não funcionais

Os requisitos não funcionais (como vimos no capítulo de especificação, relacionados com características de compatibilidade, confiabilidade, desempenho, manutenibilidade, usabilidade, segurança ou portabilidade), afetam diretamente decisões arquiteturais e comumente resultam em restrições no projeto arquitetural. Por exemplo, quando os requisitos não funcionais enfatizam *desempenho*, uma tática comum é tornar operações críticas locais e minimizar a comunicação, o que tende a levar a componentes de granularidade maior. Quando enfatizam *disponibilidade* (uma subcaracterística da confiabilidade), uma tática comum é incluir componentes redundantes e mecanismos para tolerância a falhas. Quando os requisitos não funcionais descrevem necessidades de *manutenibilidade*, uma tática é utilizar componentes de granularidade mais fina, substituíveis. Para tratar requisitos não funcionais de *segurança*, uma tática é utilizar uma arquitetura baseada em camadas com recursos críticos nas camadas interiores.

Note que as decisões arquiteturais comumente nos fazem incorrer em *trade-offs* em função dos requisitos não funcionais a serem alcançados. Por exemplo, utilizar componentes de granularidade maior tende a melhorar o desempenho, mas afeta negativamente a manutenibilidade. Introduzir componentes redundantes melhora a disponibilidade, mas dificulta o alcance da segurança. Diversos outros exemplos poderiam ser citados e exemplificam que a importância relativa entre os requisitos não funcionais e o grau com

que cada um deles precisa ser alcançado, devidamente discutido com os *stakeholders*, deve ser considerada com cautela.

De fato, um bom ou uma boa engenheira de software deve conhecer a respeito desses *trade-offs* e padrões arquiteturais e táticas que podem ser empregados para alcançar as diferentes características desejadas de um sistema de software. Esses estilos arquiteturais e táticas estão amplamente documentados para sistemas convencionais em livros sobre arquitetura de software. Para conhecer mais sobre essas táticas, recomendamos o livro *Designing Software Architectures: A Practical Approach*, de Cervantes e Kazman (2016), que tem uma lista de táticas para tratar diferentes tipos de requisitos não funcionais, além de conter exemplos de arquiteturas de referência para sistemas de software.

13.2 Considerações sobre arquitetura de sistemas de software inteligentes

Daqui em diante enfatizaremos o que *não* é encontrado na literatura convencional, que são considerações e exemplos de arquiteturas de sistemas de software inteligentes envolvendo *Machine Learning*. Lewis, Ozkaya e Xu (2021) investigaram desafios arquiteturais para sistemas que envolvem *Machine Learning*. Eles reforçaram que é preciso entender quão bem as práticas de arquitetura de software existentes suportam sistemas que envolvem componentes de *Machine Learning* e que é preciso adaptar padrões e táticas para responder a requisitos de qualidade importantes no contexto de *Machine Learning*. Em particular, eles destacam a importância de contemplar a monitoração dos modelos como um gatilho para a necessidade de manutenção e evolução do sistema e que é preciso apoiar a coarquitetura e coversionamento para alinhar sistemas e *pipelines* de desenvolvimento de modelos.

Isso é totalmente consistente com nossas experiências práticas e, neste livro, trazemos exemplos concretos de práticas de arquitetura de software aplicadas a sistemas de *Machine Learning*, considerando requisitos de qualidade importantes no contexto de *Machine Learning*. Também mostraremos como contemplar aspectos de monitoração com base em lições aprendidas de uma solução premiada (Prêmio Inventor Petrobras 2022), que foi entregue com sucesso e que teve pedido de patente depositado recentemente. O alinhamento da coarquitetura e do coversionamento será parcialmente abordado neste capítulo e parcialmente no capítulo *Gerência de configuração, DevOps e MLOps em sistemas inteligentes*.

No capítulo anterior, vimos como armazenar e implantar um modelo de *Machine Learning*. Com base nisso, podemos pensar na arquitetura da solução como um todo que utilizará um componente de *Machine Learning*. É uma boa prática pensar na arquitetura do sistema de software inteligente desde o início do desenvolvimento do projeto. Deve-se analisar, por exemplo, qual limitação a infraestrutura pode ter no seu modelo de *Machine Learning*? Quais componentes a equipe de engenharia deve construir para operacionalizar o modelo de *Machine Learning*? O modelo de *Machine Learning* precisa gerar predições off-line ou em tempo real?

Pensar nessas características com antecedência ajuda a projetar um fluxo de trabalho melhor e evita falhas na fase de produção. Por outro lado, não pensar na arquitetura e na infraestrutura pode levar ao fracasso do projeto. Por exemplo, um modelo muito grande pode ser inutilizado devido a restrições de recursos de infraestrutura para executá-lo e mantê-lo.

A arquitetura de sistemas de software inteligentes depende fortemente dos requisitos de infraestrutura e da forma de implantação pretendida. Como vimos nos primeiros capítulos deste livro sobre a especificação de sistemas com componentes de *Machine Learning*, há um conjunto de preocupações relacionadas à

infraestrutura que devem ser analisadas e, se aplicáveis, projetadas e implementadas.

Funcionalmente, a construção de sistemas de software inteligentes envolve pensar na operacionalização de componentes como a transferência dos dados desde sua origem até o modelo, o armazenamento e gestão dos artefatos de *Machine Learning*, a atualização ou retreinamento do modelo e o monitoramento dele. Ainda no contexto de arquiteturas de *Machine Learning*, é comum que sejam empregadas ferramentas interativas para visualização de dados, que permitem gerar *dashboards* (painéis de visualização). Exemplos dessas ferramentas são *Power BI*, *Spotfire* e *Tableau*. Neste caso, as predições do modelo podem ser escritas em um banco de dados para permitir gerar visualizações.

Na figura a seguir, temos um exemplo de uma arquitetura que utiliza essa forma de visualização e alguns dos demais conceitos apresentados. Nesta arquitetura, os dados, estruturados ou não estruturados e de diferentes fontes, são armazenados em um *Data Lake*. A partir desse *Data Lake* é construído o modelo de *Machine Learning*, que é então implantado na nuvem em uma função *serverless*.

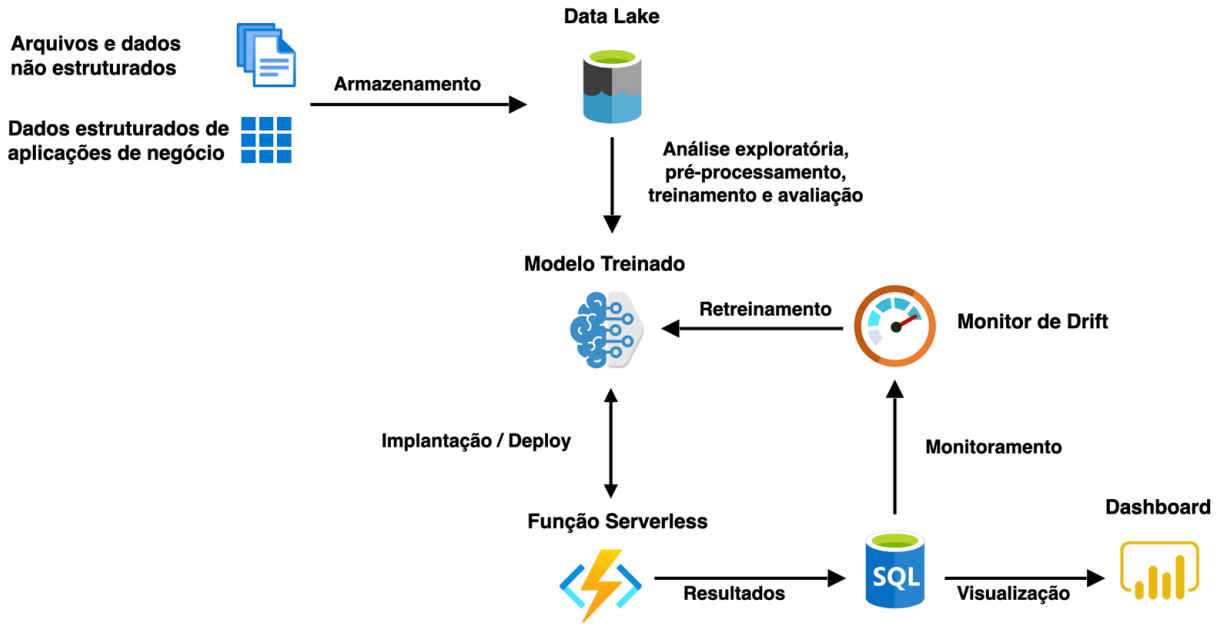


Figura 13.1: Exemplo de uma arquitetura de alto nível de um sistema de software inteligente.

Esse exemplo de arquitetura foi desenhado considerando uma função que será acionada de tempos em tempos para realizar a predição, cujos resultados são armazenados em um banco de dados, o que possibilita a visualização das predições através de um *dashboard*. A partir dos resultados armazenados, é também possível monitorar flutuações no desempenho do modelo, que podem ocorrer, por exemplo, em função de mudanças no comportamento dos dados (*concept drift*). Nesses casos, o retreinamento do modelo deve ser acionado.

O retreinamento poderá ocorrer de forma automática caso se tenha investido na automação de tarefas normalmente realizadas pelo(a) cientista de dados, como pré-processamento, treinamento e avaliação do modelo. Estes esforços de automação, incluindo a detecção de flutuações no desempenho, fazem parte do *pipeline* de MLOps, permitindo que o modelo de *Machine Learning* seja reimplantado em operação quando necessário.

13.3 Arquiteturas para diferentes tipos de predição de Machine Learning

Depois de treinar, avaliar e ajustar um modelo de *Machine Learning*, o modelo é implantado em produção para fornecer predições. Um modelo de *Machine Learning* pode fornecer predições de duas maneiras: *off-line* ou *em tempo real*.

Predição off-line

Nesse tipo de predição, o modelo de *Machine Learning* é usado para processar em lote um grande número de dados (também chamado *batch*). Nas predições em *batch*, tipicamente são usados dados históricos para gerar a predição sem a necessidade de capturar dados on-line. Esse processo pode ser executado diariamente, semanalmente ou até mensalmente, dependendo da frequência com que sejam requeridas novas predições. Exemplos de aplicação desse tipo de predição incluem:

- **Predição de demanda:** estimar a demanda de produtos por loja diariamente para otimização de estoque e entrada;
- **Análise de sentimento:** identificar o sentimento geral dos clientes em relação aos produtos de uma loja analisando o feedback do cliente.

A figura a seguir mostra uma arquitetura típica de alto nível para realizar predições em *batch*.

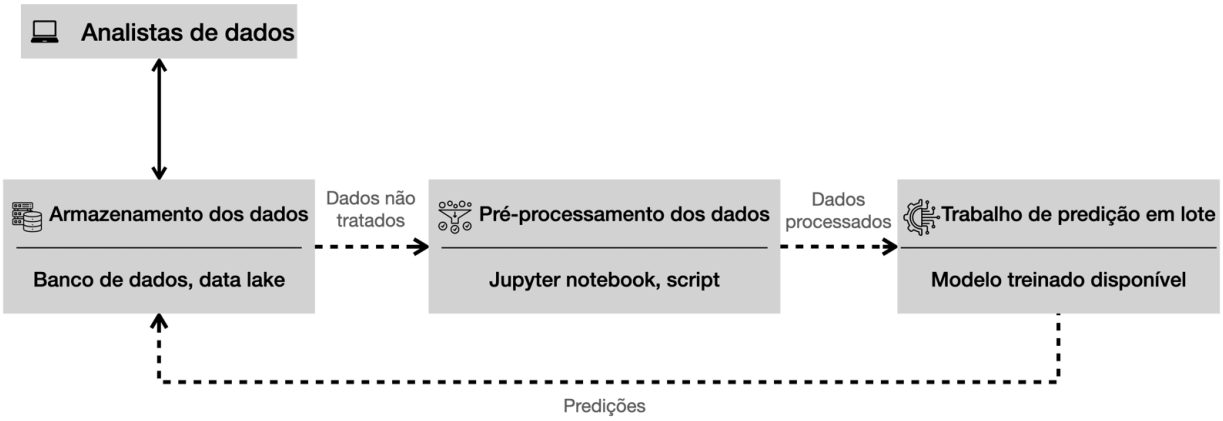


Figura 13.2: Arquitetura de predição em batch.

A primeira etapa consiste em ter acesso aos dados que serão usados para gerar as predições. Esses dados geralmente são armazenados em um banco de dados ou em um *Data Lake*. O próximo passo é pré-processar os dados envolvendo operações de limpeza nos dados, bem como a transformação dos dados em um formato adequado alinhado com os dados de treinamento do modelo de *Machine Learning*. Os dados pré-processados são inseridos no modelo, que os utiliza para fornecer predições sobre novos dados. Isso é conhecido como **predição off-line em *batch***. Por último, as predições geradas pelo modelo de *Machine Learning* são armazenadas em um banco de dados ou *Data Lake* para análise posterior ou para serem usadas por outros aplicativos.

Predição em tempo real

Nesse tipo de predição, o modelo de *Machine Learning* geralmente recebe um único ponto de dados e espera-se que forneça uma predição para esse ponto de dados em (quase) tempo real, isso com base em solicitações on-line. Ao contrário da predição off-line, nas predições em tempo real se tem uma necessidade de conhecer o contexto atual dos dados, juntamente com informações históricas, para gerar a predição. Exemplos de aplicação desse tipo de predição incluem:

- Prever se uma determinada peça de uma máquina falhará nos próximos *N* minutos, com base nos dados do sensor em tempo real;
- Estimar quanto tempo levará uma entrega de comida com base no tempo médio de entrega em uma área nos últimos 30 minutos, levando em consideração as informações do trânsito em tempo real.

As predições em tempo real podem ser entregues aos consumidores (usuários, aplicativos, painéis) de forma *síncrona* ou *assíncrona*.

Sincronicamente: Aqui a solicitação de predição e a resposta são executadas em sequência entre o chamador e o modelo de *Machine Learning*. Em outras palavras, o chamador espera até receber a predição do modelo de *Machine Learning* antes de executar as etapas subsequentes. A figura a seguir mostra uma arquitetura síncrona simples para predição on-line.

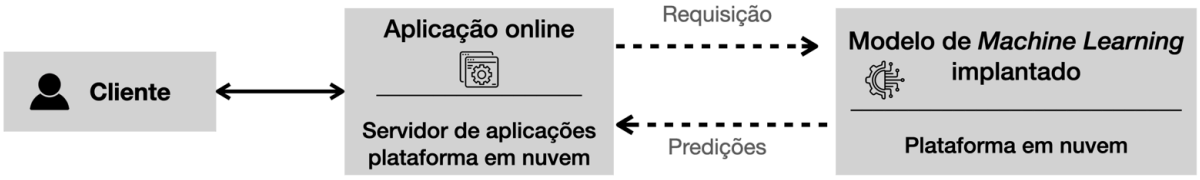


Figura 13.3: Arquitetura síncrona simples de predição on-line.

Veja que no exemplo de arquitetura de predição em tempo real temos uma aplicação on-line que envia solicitações HTTP para um modelo de *Machine Learning* que foi implantado como um microsserviço e que expõe uma API REST para predição. Na capítulo anterior, vimos como implantar um modelo de *Machine Learning* como um serviço separado na nuvem, que nada mais é um *endpoint HTTP* criado usando alguma plataforma de computação de provedores como *Google*, *Microsoft* ou *Amazon*. Após o modelo fornecer a predição, a aplicação on-line recebe a resposta.

Na maioria das vezes, é necessário implementar a lógica no *back-end* da aplicação como parte do sistema. Por exemplo, o *back-end*

da aplicação pode precisar executar o pré-processamento dos dados de entrada antes de invocar o modelo de *Machine Learning*. Da mesma forma, o *back-end* da aplicação pode precisar executar o pós-processamento na predição de saída antes de enviar a resposta de volta ao cliente.

Assincronamente: Aqui as predições ou suas ações subsequentes são entregues ao consumidor independentemente da solicitação de predição. Um exemplo são os sistemas de detecção de fraude que precisam notificar outros sistemas para que tomem providências quando uma transação potencialmente fraudulenta for identificada. A figura a seguir mostra outra arquitetura comum para predições on-line feitas de forma assíncrona.

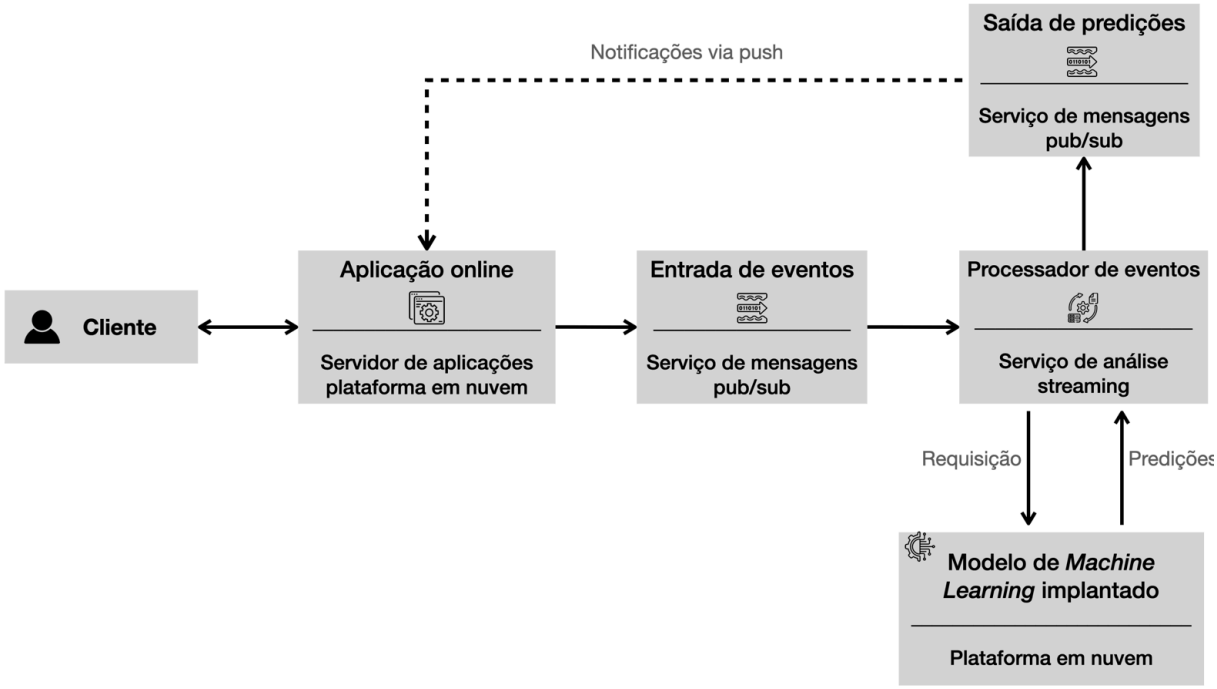


Figura 13.4: Arquitetura assíncrona de predição on-line.

Nesse exemplo de arquitetura assíncrona, o fluxo de dados funciona da seguinte maneira: a aplicação on-line envia eventos (dados que o modelo precisa para fornecer predições) para um serviço de mensagens em tempo real, conhecido como padrão *publish-subscribe (Pub/Sub)*.

Esses eventos são consumidos em tempo real e processados, caso precise, por um serviço de *streaming* de dados. Esse serviço de processamento invoca o modelo de *Machine Learning* e envia as predições para outro componente *Pub/Sub*. As mensagens do *Pub/Sub* contendo predições são enviadas de volta para a aplicação on-line e finalmente consumidas pelo cliente para monitoramento ou tomada de decisão em tempo real.

13.4 Otimização das predições dos modelos de Machine Learning

Os sistemas de software inteligentes, principalmente aqueles que envolvem predições em tempo real, têm uma propriedade importante que devem otimizar: *minimizar a latência das predições*. Essa propriedade refere-se ao tempo necessário para o modelo de *Machine Learning* processar uma unidade de dados e retornar o resultado da predição. Uma vez que o modelo de *Machine Learning* fornece a predição, espera-se que uma ação aconteça imediatamente. Isso se aplica perfeitamente para o exemplo de detecção de fraude dado para predições assíncronas, onde o tempo de resposta do modelo tem que ser muito rápido (na ordem dos milissegundos) para conseguir bloquear uma transação potencialmente fraudulenta.

Geralmente, temos duas estratégias para melhorar a latência das predições:

- **Minimizar o tempo que o modelo de *Machine Learning* leva para fazer uma predição quando é invocado.**

Podemos criar modelos de *Machine Learning* menores (menos complexos), por exemplo, reduzindo o número de camadas em uma rede neural, ou o número de níveis em uma árvore de decisão. Por outro lado, é importante refatorar o código envolvido na operacionalização do modelo (pré-processamento e pós-processamento dos dados, inferência do modelo, escrita dos resultados do modelo), removendo partes não utilizadas, redundantes ou irrelevantes. Normalmente, isso é feito ao passar da fase de experimentação para a fase de operação do modelo de *Machine Learning*. Além de otimizar o modelo em termos de complexidade e melhorias no desenho e na implantação, é possível usar aceleradores na infraestrutura para melhorar o desempenho do tempo de resposta. Normalmente, GPU (*Graphic Processing Unit*) e TPU (*Tensor Processing Unit*) são otimizadas para cargas de trabalho de *Machine Learning*. No entanto, o uso desses recursos de infraestrutura implica um custo adicional que deve ser analisado, conforme mostrado na seção de especificação de sistemas de software inteligentes.

- **Minimizar o tempo que o sistema leva para fornecer a predição assim que recebe a solicitação.** Inclui armazenar as entradas do modelo de *Machine Learning* em um repositório de dados de consulta com baixa latência de leitura, pré-computar predições em um trabalho off-line em *batch* e armazenar as predições em cache. Apenas otimizar o modelo de *Machine Learning* para otimizar a latência não garante uma redução na latência geral do sistema ao implantar o modelo de *Machine Learning* em produção. Lembremos que, para que um modelo de *Machine Learning* forneça uma predição, o ponto de dados precisa incluir todas as variáveis usadas para treinar o modelo. Por exemplo, ao treinar um modelo para estimar a probabilidade de manifestação de uma comunidade localizada ao redor de uma refinaria de petróleo, segundo dados meteorológicos como velocidade e direção do vento, umidade, temperatura e variáveis de processo da refinaria, o modelo de *Machine Learning* treinado precisa dos valores dessas variáveis como entradas para calcular a estimativa. No entanto, em

muitos casos, esses valores não são fornecidos pelo acionador do modelo de *Machine Learning*. Em vez disso, eles são lidos em tempo real a partir de um repositório de dados ou calculados no *back-end* da aplicação. Essa capacidade de buscar com rapidez suficiente os dados que o modelo precisa, além de computar e manter valores em tempo real para serem usados como novas entradas para o modelo, constituem um desafio para que um modelo de *Machine Learning* forneça predições de baixa latência.

Até aqui já vimos como otimizar os modelos de *Machine Learning* para o fornecimento das predições, mas nem sempre essa otimização envolve decisões simples. Por exemplo, quanto mais complexo for o modelo de *Machine Learning*, maior será a capacidade do modelo de capturar relações nos dados e, portanto, melhor tenderá a ser a eficácia preditiva (supondo que o *overfitting* é tratado com cuidado). No entanto, quanto maior o modelo de *Machine Learning*, mais tempo ele tenderá a levar para produzir uma predição. Usar um modelo de *Machine Learning* menos complexo, por outro lado, reduz a latência de predição, mas pode não ter a eficácia preditiva de um modelo maior.

Portanto, há uma compensação (*trade-off* de *Machine Learning*), que deve ser tratada desde a especificação do sistema de software inteligente, entre a eficácia preditiva e a latência de predição do modelo de *Machine Learning*. Dependendo dos requisitos definidos, dois aspectos devem ser levados em consideração:

1. **A eficácia preditiva do modelo de *Machine Learning* em termos das métricas definidas**. Exemplos dessas métricas incluem acurácia e precisão para modelos de classificação e *R2 score* e erro quadrático médio para modelos de regressão.
2. **A restrição operacional a ser atendida pelo modelo de *Machine Learning***, como a latência da predição. Por exemplo, definir um valor específico (100 milissegundos) como limite para fornecer uma predição. Outro exemplo é o tamanho do modelo,

que pode ter restrições – em dispositivos móveis ou embarcados, por exemplo.

Neste capítulo, apresentamos exemplos de arquiteturas que temos utilizado para sistemas de software inteligentes envolvendo *Machine Learning*, tanto para predições off-line quanto para predições em tempo real, e fornecemos recomendações para otimizar as predições. Consideramos fundamental que a arquitetura seja pensada no início do projeto com base nos requisitos identificados, por exemplo, aplicando a técnica PerSpecML (vista no capítulo sobre Especificação de sistemas de software inteligentes) para ajudar a obter uma visão geral dos requisitos relevantes a serem considerados. No capítulo seguinte, abordaremos o projeto detalhado, fornecendo boas práticas mais próximas da codificação da solução.

CAPÍTULO 14
Projeto de sistemas de software inteligentes

Neste capítulo, veremos conceitos básicos sobre projeto de software e discutiremos como eles podem ser aplicados no contexto de sistemas de software inteligentes envolvendo *Machine Learning*, com exemplos da aplicação dos populares princípios SOLID. Além dos princípios, que são mais genéricos, norteadores e de utilidade geral, faremos considerações sobre a aplicabilidade de padrões de projeto, *code smells* e refatorações.

14.1 Conceitos básicos de projeto de software

O **projeto de software** representa a busca de uma solução para o problema descrito nos requisitos. Enquanto a especificação de requisitos descreve (de forma abstrata para requisitos de usuário ou em detalhes para requisitos do sistema) **o que precisa ser feito**, o projeto descreve **como deve ser feito**. Essa divisão, muitas vezes mal compreendida, traça a linha entre o que se entende como requisitos e como projeto de software. Enquanto os requisitos são normalmente descritos pelo(a) analista de requisitos ou pelo(a) *Product Owner*, o projeto envolve a equipe técnica e profissionais focados na solução técnica, como arquitetos de software ou desenvolvedores.

Enquanto a arquitetura de software – sobre a qual falamos no capítulo anterior – enfatiza a documentação de decisões de projeto de alto nível, em geral referentes ao sistema como um todo, o projeto de software diz respeito às decisões mais próximas ao código e fortemente relacionadas com os requisitos funcionais.

Embora exista uma linguagem estabelecida como universal para desenhar diagramas e modelar sistemas orientados a objetos, a UML (*Unified Modeling Language*), o seu uso na prática é limitado. A pesquisa de Petre (2013) identificou que a maioria dos profissionais simplesmente não utiliza UML ou a utiliza de forma seletiva e informal. No contexto nacional, o cenário é parecido: em um *survey* recente na indústria brasileira com 314 profissionais, cerca de 75% informaram não utilizar UML na prática (WILSON JÚNIOR *et al.*, 2021). Em projetos de Ciência de Dados, esse uso tende a ser ainda mais restrito. Na prática, embora a UML possa apoiar atividades de documentação e manutenção de sistemas complexos (FERNÁNDEZ-SÁEZ *et al.*, 2018), ela acaba sendo mais usada para fins de comunicação e para raciocinar a respeito de ideias de projeto (PETRE, 2013).

Em seu livro *UML Essencial*, Fowler (2014) considera três formas de uso da UML: como *blueprint*, como linguagem de programação e como esboço. O uso da UML como *blueprint*, que corresponde a elaborar todo o projeto antes da implementação, como se fosse uma planta técnica, é pouco usual em contextos ágeis. Em relação ao seu uso como linguagem de programação, para a geração de código, sua utilidade para esse propósito é alvo de debates. Em geral, os desenvolvedores se sentem mais confortáveis escrevendo código-fonte em uma IDE do que desenhando diagramas ultradetalhados (para gerar código) em uma ferramenta CASE. Por fim, o uso como esboço corresponde a construir diagramas leves e mais informais que podem ser utilizados para fins de comunicação e para discutir ideias de projeto.

Com base em experiências no desenvolvimento de sistemas de software inteligentes, recomendamos que a UML seja utilizada para *esboçar a solução* em projetos de Ciência de Dados, principalmente se esses projetos forem conduzidos seguindo métodos ágeis. É também dessa forma que a utilizaremos ao longo deste capítulo, para explicar ideias de projeto, sem enfatizar detalhes de notação (embora as notações estejam corretas e os exemplos apresentem os

códigos relacionados). Recomendamos que profissionais envolvidos na construção de software conheçam a UML ou ao menos o diagrama de classes, que contém elementos estruturais equivalentes aos de uma linguagem de programação orientada a objetos. Mais informações sobre a UML podem ser encontradas no livro *UML Essencial* (FOWLER, 2014).

Reforçamos que a UML é apenas uma notação para diagramação orientada a objetos. Existem diversas ferramentas CASE que permitem desenhar diagramas UML. Os diagramas deste capítulo, por exemplo, foram desenhados na ferramenta *Visual Paradigm Community Edition*. A UML como linguagem é trivial e relativamente pouco importante quando não aliada a princípios que promovam seu bom uso. Ou seja, saber como ler e desenhar diagramas UML é parte das habilidades básicas de um bom desenvolvedor ou boa desenvolvedora, mas é apenas um primeiro passo para elaborar bons projetos orientados a objetos. Mais importante do que saber desenhar os diagramas UML ou conhecer ferramentas CASE são as habilidades de projeto de sistemas orientados a objetos.

Em um sistema orientado a objetos, uma habilidade crítica é projetar e pensar em **classes** (e consequentemente os *objetos* que serão criados a partir delas). A boa notícia é que isso pode ser aprendido com base em princípios explicáveis. O projeto de um sistema orientado a objetos normalmente é feito do ponto de vista da seguinte metáfora:

- Objetos possuem responsabilidades;
- Objetos colaboram.

Assim, duas preocupações básicas são a definição das responsabilidades de cada objeto e identificar com quais outros objetos ele colabora. Um conceito que confunde esse entendimento é o de *responsabilidade*. Uma responsabilidade (ou *concern*) – por exemplo, determinar o valor do frete em um sistema de comércio eletrônico, ou realizar um pré-processamento de dados para um componente de *Machine Learning* – pode ser implementada na

prática através de um conjunto de métodos relacionados, idealmente de uma mesma classe. Conceitos essenciais que nos auxiliam nessa distribuição de responsabilidades são os princípios do **especialista da informação**, da **alta coesão** e do **baixo acoplamento** (LARMAN, 2007).

O **especialista da informação** é o princípio geral mais básico a respeito da atribuição de responsabilidades. Ele sugere que se atribua uma responsabilidade à classe que tenha a (maior parte da) informação necessária para satisfazê-la. Quem tem a informação realiza o trabalho! Ou seja, na prática, os métodos devem ficar na mesma classe em que se encontram os dados que eles manipulam. Assim, para determinar que classe do software deve realizar determinada responsabilidade, nos perguntamos primeiro que informação é necessária para realizar essa responsabilidade e buscamos a classe que tem a maior parte dessa informação.

A **alta coesão** busca manter as classes focadas, inteligíveis e gerenciáveis e, como efeito colateral, ainda apoia o baixo acoplamento. Coesão mede informalmente quão relacionadas funcionalmente são as operações de uma classe. O princípio sugere atribuir responsabilidades às classes de modo que a coesão permaneça sempre alta. Idealmente, cada classe possui apenas uma responsabilidade dentro do sistema. Por essa razão, a alta coesão é também frequentemente tratada através do nome *separation of concerns* (separação de interesses), que envolve separar as responsabilidades (*concerns*, ou interesses) em diferentes classes, de modo que cada classe tenha somente uma responsabilidade. Cabe ressaltar que uma mesma responsabilidade pode ser implementada através de diversos métodos. A alta coesão é percebida na prática por desenvolvedores de software como um dos aspectos mais críticos para facilitar a manutenção de sistemas de software (FERNANDES; KALINOWSKI, 2023).

O **baixo acoplamento** visa reduzir o impacto de modificações, apoiando diretamente a manutenibilidade. O princípio sugere atribuir responsabilidades de modo que O acoplamento (relacionamentos)

entre as classes permaneça baixo. Acoplamento baixo tende a reduzir o esforço e defeitos na modificação de software, assegurando que as modificações em uma classe afetem poucas outras classes do sistema. Desta forma, os efeitos dessas modificações podem ser contidos alterando poucas outras classes. Esse princípio é utilizado tanto para escolher entre novas alternativas como para avaliar projetos existentes. Se todas as outras coisas permanecerem iguais, devemos optar por um projeto com um acoplamento mais baixo em relação às alternativas. A alta coesão acaba apoiando o baixo acoplamento: ao agruparmos operações relacionadas a uma mesma responsabilidade em uma só classe, acabamos reduzindo a necessidade de comunicação entre classes para realizar a responsabilidade em questão.

Esses princípios são essenciais e deveriam ser conhecidos por todos que de alguma forma trabalham desenvolvendo software. Eles formam a base para entender outros princípios para um bom projeto de software. Entre os princípios de orientação a objetos mais amplamente difundidos estão os princípios SOLID, cujos conceitos foram apresentados no capítulo de boas práticas de projeto e construção de sistemas. Daqui em diante veremos exemplos de como eles podem ser aplicados no contexto de *Machine Learning*.

14.2 Princípios SOLID aplicados a Machine Learning

Assim como a construção de qualquer tipo de código, o código de *Machine Learning* pode se beneficiar da aplicação de princípios de projeto para permitir a construção de códigos que sejam mais fáceis de entender e manter, que tenham boa qualidade estrutural. Para exemplificar, falaremos dos princípios SOLID, que são amplamente conhecidos e utilizados no mercado para auxiliar a construção de software orientado a objetos, e que já foram detalhados nos

capítulos anteriores. Relembrando, os princípios SOLID buscam ajudar a realizar um bom projeto orientado a objetos para que um software tenha código flexível, escalável, sustentável e reutilizável. SOLID é um acrônimo para cinco princípios:

- ***Single Responsibility Principle* (SPR)**: Princípio de Responsabilidade Única
- ***Open/Closed Principle* (OCP)**: Princípio Aberto / Fechado
- ***Liskov Substitution Principle* (LSP)**: Princípio da Substituição de Liskov
- ***Interface Segregation Principle* (ISP)**: Princípio de Segregação de Interface
- ***Dependency Inversion Principle* (DIP)**: Princípio da Inversão de Dependência

A seguir, retornaremos a cada um desses princípios com exemplos de aplicação para *Machine Learning*. Em alguns dos exemplos, utilizaremos códigos escritos para a biblioteca Scikit-learn.

Princípio de Responsabilidade Única (SRP)

Esse princípio diz que uma classe não deve ter mais de um motivo para ser alterada, ou seja, uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software. Quando uma classe tem mais de uma responsabilidade e surge uma necessidade de alteração, será difícil modificar uma dessas responsabilidades sem comprometer as outras. Violar esse princípio pode trazer problemas como alto acoplamento, baixa coesão, dificuldades na implementação de testes automatizados e de reaproveitamento do código.

Vejamos um exemplo no contexto de *Machine Learning*. A construção de modelos de *Machine Learning* normalmente exige um conjunto de atividades a serem realizadas, entre elas a obtenção dos dados de diferentes fontes, o pré-processamento, a construção do modelo e a avaliação do modelo. Essas atividades podem ser mapeadas em diferentes responsabilidades dentro de um sistema a

ser construído. Assim, conforme ilustrado na figura a seguir, poderíamos pensar em ao menos quatro classes com responsabilidades bem definidas e distintas.

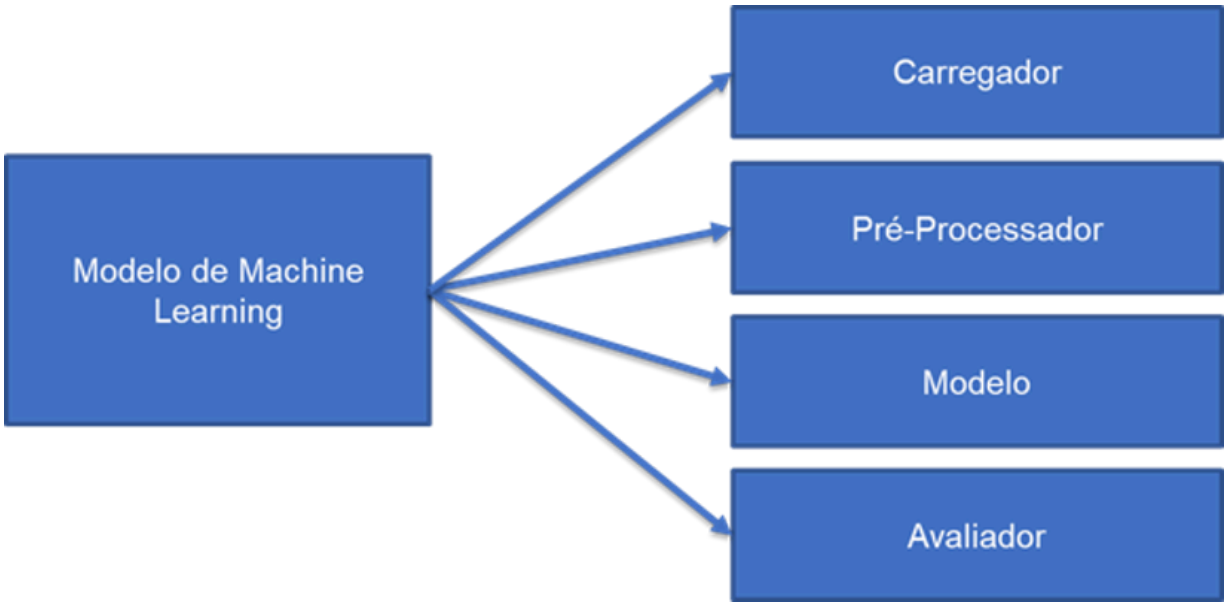


Figura 14.1: 4 classes com responsabilidades bem definidas e distintas.

Vamos ver alguns códigos de exemplo comentados. Note que os códigos foram propositalmente simplificados para que possamos focar em aspectos estruturais na nossa discussão. O código a seguir apresenta a classe `Carregador`. Ela possui um único método utilitário, chamado `carregar_dados`. Basicamente o que ela faz é internamente chamar o comando `read_csv` da biblioteca Pandas, retornando um *DataFrame* do Pandas com todos os dados carregados. Claro que essa classe poderia precisar de outros métodos caso a carga de dados fosse proveniente de diferentes fontes, por exemplo. De qualquer forma, essa responsabilidade referente à carga dos dados seria tratada dentro dessa classe.

```
class Carregador:

    def carregar_dados(self, url: str, atributos: list):
        """ Carrega e retorna um DataFrame. Há diversos parâmetros no
read_csv que poderiam ser utilizados para dar opções adicionais."""
        return pd.read_csv(url, names=atributos)
```

A classe `PreProcessador`, por sua vez, trata da responsabilidade de pré-processamento. Ela possui um método `pre_processar`, que é responsável pela limpeza dos dados, seleção de atributos (*feature selection*), a divisão dos dados em treino e teste e a normalização ou padronização dos dados. O código encontra-se a seguir.

```
class PreProcessador:

    def pre_processar(self, dataset, percentual_teste, seed=7):
        """ Cuida de todo o pré-processamento. """
        # limpeza dos dados e eliminação de outliers

        # feature selection

        # divisão em treino e teste
        X_train, X_test, Y_train, Y_test =
            self.__preparar_holdout(dataset, percentual_teste, seed)
        # normalização/padronização

        return (X_train, X_test, Y_train, Y_test)

    def __preparar_holdout(self, dataset, percentual_teste, seed):
        """ Divide os dados em treino e teste usando o método holdout.
        Assume que a variável target está na última coluna.
        O parâmetro test_size é o percentual de dados de teste.
        """
        dados = dataset.values
        X = dados[:, 0:-1]
        Y = dados[:, -1]
        return train_test_split(X, Y, test_size=percentual_teste,
random_state=seed)
```

Para fins de simplificação, a limpeza dos dados e eliminação de *outliers* não está implementada. Normalmente essa limpeza envolve tratar dados faltantes (por exemplo, substituindo-os pela mediana do restante dos dados, quando pertinente) e eliminar dados que desviam significativamente do comportamento esperado (*outliers*) e que poderiam enviesar o comportamento do modelo a ser construído. O pré-processamento conceitualmente envolve também

a seleção dos atributos (*feature selection*) a serem considerados para a construção do modelo, mantendo somente os atributos que realmente ajudam a construir um modelo que permita boas predições.

Outro aspecto importante do pré-processamento é a divisão dos dados em treino e teste. Para tratar esse aspecto, no código foi criada uma função não pública chamada `__preparar_holdout`, que não é visível externamente (em Python, os métodos não públicos iniciam com dois underscores, `__`), mas que apoia a separação dos dados de teste. Basicamente, o que essa função faz é assumir que a variável-alvo está na última coluna, dividir os dados em *X* (entradas) e *Y* (saída) e acionar a função `train_test_split` do Scikit-learn, que separa os dados *X* e *Y* em *X* de treino e teste e *Y* de treino e teste, respeitando um percentual fornecido como parâmetro.

Por fim, o pré-processamento poderia ainda aplicar ações de normalização ou padronização dos dados, para então retornar os conjuntos `X_train`, `X_test`, `Y_train` e `Y_test` já com os devidos tratamentos para que possam ser utilizados na construção do modelo. É importante ressaltar que esse é apenas um exemplo simplificado e que as ações concretas a serem realizadas durante o pré-processamento podem ser definidas com base no conhecimento das técnicas e experimentação para ver quais técnicas efetivamente ajudam a construir modelos melhores. A aplicação do princípio SRP garante que, caso se precise alterar o pré-processamento, o desenvolvedor saberá que essa responsabilidade está sendo tratada dentro desta classe, tendo um ponto único para modificação dentro do sistema.

O trecho de código a seguir mostra a classe `Modelo`, que tem a responsabilidade da construção do modelo. Nesta implementação, ela possui um único método, chamado `treinar_SVM`, que instancia a classe `SVC` da biblioteca Scikit-learn (que implementa o algoritmo SVM para classificação), treina o modelo com os dados de treino de entrada e de saída (chamando o método `fit` da classe SVC) e

retorna o modelo treinado para que ele possa ser utilizado para predições.

```
class Modelo:

    def treinar_SVM(self, X_train, Y_train):
        """ Cria e treina um modelo SVM. Poderia ter um Grid Search com
cross_validation para escolher os melhores hiperparâmetros etc."""
        modelo = SVC()
        modelo.fit(X_train, Y_train)
        return modelo
```

Novamente fornecemos uma implementação simplificada para que possamos focar didaticamente na estrutura. Existem diversos recursos que podem ser empregados durante o treinamento do modelo, como a validação cruzada e a busca dos melhores hiperparâmetros, por exemplo, utilizando *Grid Search*. O SVM tem um parâmetro que poderia ser fornecido na sua instanciação, que é a *rigidez da margem* (parâmetro *C*). Quando instanciamos a classe SVC sem fornecer nenhum parâmetro, ele assume o valor default *C=1*. Os códigos referentes à otimização de hiperparâmetros poderiam ficar dentro dessa mesma classe, que tem como responsabilidade realizar o treinamento e construção do modelo.

Por fim, a classe `Avaliador`, exibida no trecho de código a seguir, cuida da avaliação do modelo utilizando os dados de teste. Basicamente, ela invoca a predição do modelo para os dados de entrada de teste (`X_test`), compara os resultados obtidos com os dados de saída de teste (`Y_test`) e retorna a acurácia (percentual de acertos). Assim como nos outros exemplos, aqui poderiam ser utilizadas diferentes formas e métricas de avaliação, mas a responsabilidade estaria contida nesta classe.

```
class Avaliador:

    def avaliar_acuracia(self, modelo, X_test, Y_test):
        """ Faz uma predição e avalia o modelo. Poderia parametrizar o
tipo de avaliação, entre outros."""
```

```
        predicoes = modelo.predict(X_test)
        return accuracy_score(Y_test, predicoes)
```

Uma vez construídas essas classes utilitárias com responsabilidades claramente definidas, elas podem ser instanciadas como no código a seguir.

```
# Instanciação das Classes
carregador = Carregador()
pre_processador = PreProcessador()
modelo = Modelo()
avaliador = Avaliador()
```

Utilizando as classes criadas, o código para realizar as responsabilidades de carga de dados, pré-processamento, treinamento e avaliação do modelo se torna extremamente simples, conforme ilustrado a seguir.

```
# Carga
dataset = carregador.carregar_dados(url_dados, atributos)
# Pré-processamento
X_train, X_test, Y_train, Y_test = pre_processador.pre_processar(dataset,
percentual_teste)
# Treinamento do modelo
model = modelo.treinar_SVM(X_train, Y_train)
# Impressão do resultado da avaliação
print(avaliador.avaliar_acuracia(model, X_test, Y_test))
```

É possível perceber que, aplicando o princípio SRP, caso adaptações se mostrem necessárias em alguma das responsabilidades, há um ponto de modificação único e de fácil localização (na classe correspondente à responsabilidade). Além disso, melhorias realizadas em uma responsabilidade automaticamente afetarão todas as partes do código que fazem uso da classe correspondente.

Princípio Aberto/Fechado (OCP)

Esse princípio diz que uma classe deve estar aberta para extensão, porém fechada para modificação. Quando novos comportamentos e

recursos precisam ser adicionados no software, devemos estender e não alterar o código original. Por exemplo, organizar o seu código com uma superclasse genérica e abstrata o suficiente permitirá que o programa seja estendido com a adição de novas subclasses sem precisar alterar as classes existentes.

Diversas aplicações desse princípio podem ser encontradas na própria biblioteca Scikit-learn. Por exemplo, temos um conjunto de classes para tratar diferentes transformações de dados durante o pré-processamento, tais como normalização (classe `MinMaxScaler`) e padronização (classe `StandardScaler`). Se todas as diferentes formas de transformação estivessem implementadas em uma única classe, ela teria que ser modificada toda vez que se quisesse acrescentar uma nova forma de transformação de dados à biblioteca.

A solução adotada no Scikit-learn foi, de acordo com o princípio OCP, criar uma superclasse mais genérica para permitir que a biblioteca pudesse ser estendida pelo acréscimo de novas subclasses, sem precisar alterar as classes existentes. Essa superclasse mais genérica é a classe `TransformerMixin`. Novas classes que buscam realizar transformações sobre dados (assim como a `MinMaxScaler` e a `StandardScaler`) devem herdar dessa classe e implementar os métodos `fit` e `transform`. Isso permitirá a todo o restante do Scikit-learn tratar a nova classe de forma genérica como um `TransformerMixin` (uma classe que identifica todos os transformadores de dados).

A figura a seguir ilustra um *diagrama de classes* em UML (*Unified Modeling Language*, a linguagem para modelagem de programas orientados a objetos) e mostra como três diferentes tipos de transformadores (`StandardScaler`, `MinMaxScaler` e `RobustScaler`) foram criados no Scikit-learn herdando das classes mais genéricas `BaseEstimator` e `TransformerMixin`. Basicamente, o diagrama ilustra as classes mais genéricas (apontadas pelo triângulo) e as que herdam delas e proveem implementações mais específicas, além de indicar os métodos providos pelas classes. Para simplificar o entendimento, abstraia a classe `BaseEstimator` que é uma classe-base da qual todos

os estimadores do Scikit-learn herdam, assegurando que sejam capazes de listar todos os seus parâmetros de classe.

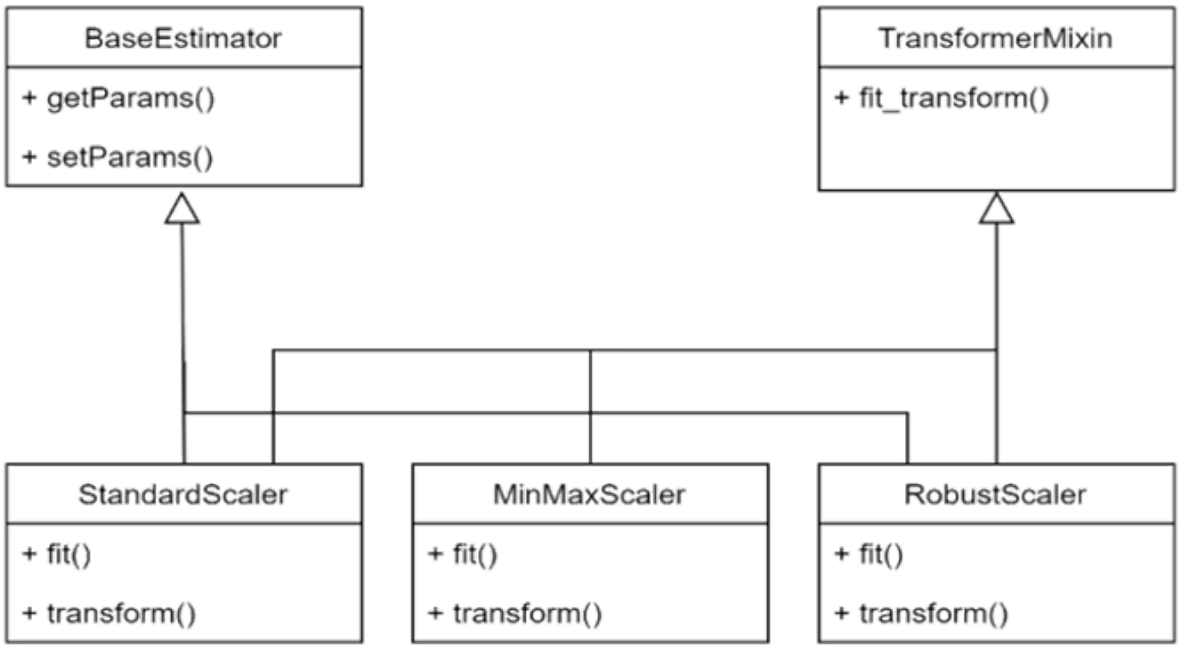


Figura 14.2: Diagrama de classes de transformadores.

O Princípio Aberto/Fechado (OCP) se manifesta nesse contexto ao assegurar que novos *Scalers* possam ser criados em classes completamente novas, sem afetar o código das demais. Para isso, basta criar uma nova subclasse que também herde de `TransformerMixin` (e de `BaseEstimator`, como todo estimador do Scikit-learn) e que implemente os métodos `fit` e `transform`. Para o restante do sistema, objetos da nova classe serão utilizados de forma transparente, como se fossem objetos da classe `TransformerMixin`, e nenhuma outra linha de código terá que ser alterada.

Ou seja, a biblioteca está fechada para modificação (o código anteriormente escrito não precisa ser alterado), mas aberta para extensão. Novas funcionalidades podem ser acrescentadas criando novas subclasses que terão integração automática no restante do código. Note que, para isso, o restante do código deverá ser

construído para depender das abstrações (as classes-base da herança) e não das implementações específicas (as subclasses). Programar dependendo das abstrações consiste no Princípio de Inversão de Dependência (DIP), que veremos mais adiante.

Princípio da Substituição de Liskov (LSP)

Esse princípio diz que uma subclasse deve poder ser substituída pela sua superclasse. Parece estar invertido, uma vez que sabemos que, se *S* é uma subclasse de *C*, então os objetos do tipo *C* podem ser substituídos pelos objetos de tipo *S* em tempo de execução. Mas, quando o LSP é seguido, os métodos públicos definidos nas subclasses devem estar também presentes na superclasse, mesmo que de forma abstrata (o que também é conhecido como *strong behavioral subtyping*).

No fundo, o LSP encoraja o uso das classes de nível de abstração mais alto, estruturando as heranças do código de forma a ampliar as possibilidades de aplicação do Princípio de Inversão de Dependência (DIP), uma vez que as abstrações (superclasses) terão os métodos necessários para expressar toda a semântica pretendida.

É importante que saibamos quando vale a pena investir nesse princípio. Repare que, no exemplo da figura anterior, o princípio LSP não foi seguido, uma vez que a classe `TransformerMixin` não tem os métodos `fit` e `transform`, que aparecem somente nas subclasses. Essa decisão provavelmente foi tomada com base em alguma certeza de que o uso pretendido da superclasse `TransformerMixin` de forma abstrata fosse restrito ao método `fit_transform`, que, pela definição de herança, é herdado também pelas subclasses.

Princípio da Segregação de Interfaces (ISP)

Esse princípio define que várias interfaces específicas são melhores do que uma interface genérica, ou seja, que uma classe não deve conhecer nem depender de métodos dos quais não necessite.

Interfaces devem ser segregadas com base nos requisitos, em vez de fornecer interfaces grandes de uso genérico.

No exemplo da figura a seguir, representado em UML, a interface `Recomendador` possui dois métodos e três classes que implementam a interface, um recomendador que utiliza filtragem colaborativa, um que utiliza *deep learning* e um que utiliza o algoritmo KNN. É possível observar que o recomendador KNN não possui uma implementação para obter recomendações personalizadas, que seriam obtidas com base em padrões comportamentais observados dos usuários. Ele se limita a identificar os itens mais próximos de um determinado item de interesse. Entretanto, como ele implementa a interface, ele teria que criar um método para obter recomendações personalizadas, mesmo que sem lógica contida, já que isso está além de sua utilização pretendida.

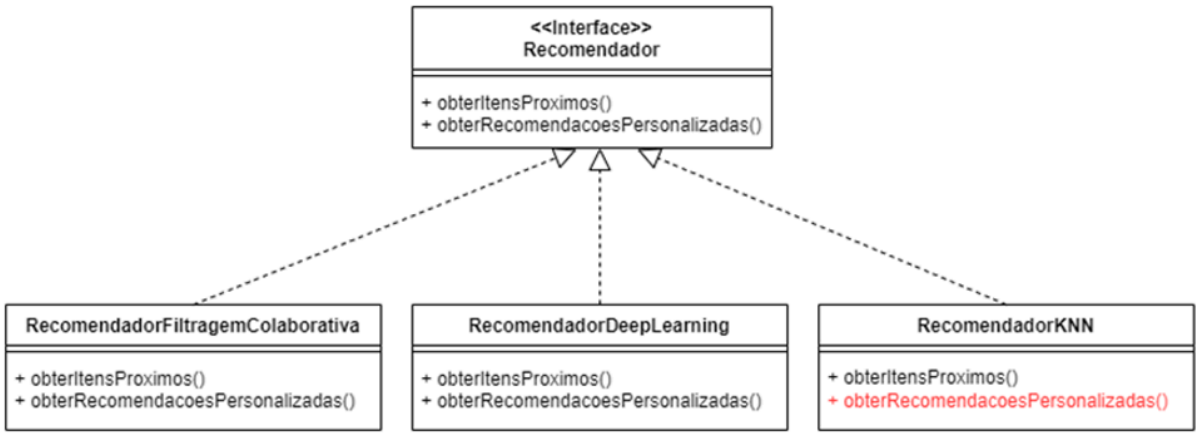


Figura 14.3: Interface Recomendador.

Nesse caso, seguindo o Princípio de Segregação de Interfaces (ISP), seria mais apropriado criar duas interfaces para que cada uma das alternativas de recomendação implemente exatamente as interfaces pretendidas. Isso evitaria que alguém viesse a utilizar o KNN e invocasse o método `obterRecomendacoesPersonalizadas`, comportamento que ele não implementa. A solução está ilustrada na figura a seguir.

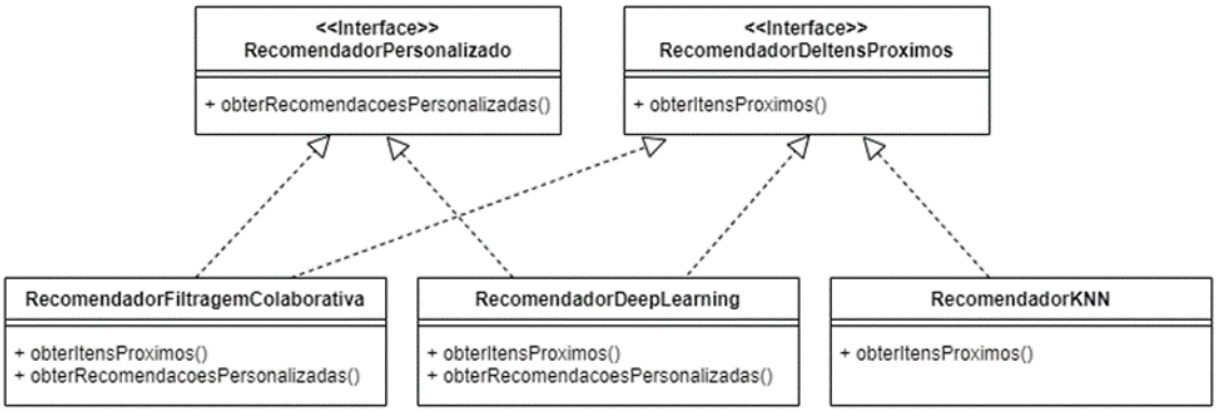


Figura 14.4: Segmentação da interface Recomendador.

O trecho de código complementa a solução e representa a implementação do diagrama UML em Python. Para fins de simplificação, apenas as classes `RecomendadorFiltragemColaborativa` e `RecomendadorKNN` foram implementadas. É possível notar que o `RecomendadorFiltragemColaborativa` implementa as duas interfaces e que, portanto, pode ser utilizado tanto como `RecomendadorPersonalizado` quanto como `RecomendadorDeItensProximos`. É possível observar ainda que não será possível utilizar o `RecomendadorKNN` como um recomendador personalizado, o que, nesse caso, é a solução apropriada.

```
from abc import ABC, abstractmethod

class RecomendadorPersonalizado(ABC):
    @abstractmethod
    def obterRecomendacoesPersonalizadas(self, user):
        pass


class RecomendadorDeItensProximos(ABC):
    @abstractmethod
    def obterItensProximos(self, item):
        pass


class RecomendadorFiltragemColaborativa(RecomendadorPersonalizado,
```

```
RecomendadorDeItensProximos):

    def obterRecomendacoesPersonalizadas(self, user):
        print("Recomendações Personalizadas.")

    def obterItensProximos(self, item):
        print("Recomendação de itens próximos.")


class RecomendadorFiltragemColaborativa(RecomendadorDeItensProximos):

    def obterItensProximos(self, item):
        print("Recomendação de itens próximos.")
```

Princípio da Inversão de Dependências (DIP)

Esse princípio define que devemos depender de abstrações e não de implementações. Programar para uma interface (a superclasse de maior abstração), e não para uma implementação (subclasse), permitirá desacoplar classes clientes de implementações específicas.

Um exemplo de aplicação do Princípio de Inversão de Dependência (DIP) da biblioteca Scikit-learn é a classe `Pipeline`, que permite concatenar um conjunto de transformadores, desde que eles sigam uma interface predefinida. O *pipeline* recebe tuplas de nomes e transformadores (`name`, `transform`) e sequencialmente aplica os transformadores e um estimador final. Os elementos `transform` das tuplas intermediárias do *pipeline* precisam implementar os métodos `fit` e `transform`. O elemento `transform` da tupla final precisa implementar somente o método `fit`.

O código a seguir mostra a construção de um *pipeline* que aplica sequencialmente uma padronização nos dados e, depois, treina o modelo KNN com os dados já padronizados e imprime os resultados de acurácia obtidos com a predição dos dados de teste. Note que o

pipeline foi construído com as tuplas `('StandardScaler', StandardScaler())` e `('KNN', KNeighborsClassifier())`, que contêm respectivamente instâncias das classes `StandardScaler` e `KNeighborsClassifier`. Ambas as classes possuem os métodos `fit` e `transform`.

Como a classe `Pipeline` é implementada considerando apenas a abstração, seria possível montar *pipelines* com outros pré-processamentos e outros tipos de modelo sem muita dificuldade.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

model = Pipeline([('StandardScaler', StandardScaler()),
                  ('KNN', KNeighborsClassifier())])

model.fit(X_train, Y_train)
accuracy_score(model.predict(X_test), Y_test)
```

Embora seja um bom exemplo do Princípio de Inversão de Dependência (DIP), já que envolve programar para a abstração, a classe `Pipeline` verifica internamente se os elementos que pretende utilizar possuem os métodos `fit` e `transform` (ou seja, se a abstração está sendo atendida) e alerta no caso de problemas. Assim, essa classe não utiliza explicitamente o conceito de interface. Provavelmente essa decisão de projeto foi tomada considerando que as exigências para o seu último elemento são diferentes das exigências para os demais.

Você pode acessar o código-fonte desses exemplos no link https://colab.research.google.com/github/profkalinowski/livroescd/blob/main/livro_ESCD_projeto_de_sistemas.ipynb.

14.3 Padrões de projeto aplicados a Machine Learning

Os padrões de projeto foram introduzidos por Gamma *et al.* (1995). Enquanto os princípios até então discutidos são mais gerais e norteadores, os **padrões de projeto**, em geral, apresentam ideias mais avançadas envolvendo soluções padronizadas que apresentaram sucesso na solução de problemas recorrentes no projeto de sistemas de software. Para isso, os padrões documentam não só a solução, mas também o problema a ser resolvido – ou seja, se baseiam na ideia intuitiva de que padrões de solução advêm de padrões nos problemas.

De maneira simplificada, os padrões de projeto podem ser vistos como organizações entre classes e a forma com que essas classes interagem para resolver um problema recorrente. Embora eles se apliquem ao projeto, comumente apresentam detalhes em código. Eles permitem transmitir conhecimento sobre o projeto de sistemas e aprimorar a comunicação entre desenvolvedores.

Embora muitos desses padrões tenham sido originalmente concebidos para o desenvolvimento de software convencional, eles também podem ser aplicados na construção de sistemas de *Machine Learning*, provendo soluções elegantes. Enquanto a aplicação dos padrões se dá de maneira similar a como ela ocorre em sistemas convencionais, forneceremos exemplos de soluções em que sistemas de *Machine Learning* se beneficiam significativamente da aplicação de padrões de projeto, reforçando a importância do estudo da Engenharia de Software para se tornar uma pessoa engenheira de *Machine Learning* diferenciada.

O padrão ***Strategy*** é um desses exemplos. Esse é um dos padrões originalmente propostos por Gamma *et al.* (1995). Ele é usado para permitir que diferentes estratégias (algoritmos) sejam usadas em tempo de execução em diferentes cenários ou contextos. Isso pode ser extremamente útil em sistemas inteligentes, permitindo

facilmente trocar – em tempo de execução e em função de características de contexto da sua escolha – entre diferentes algoritmos de *Machine Learning*. Como os padrões de projeto estão bem documentados na literatura de Engenharia de Software, forneceremos os detalhes de implementação apenas para este, visando exemplificar a aplicação do padrão a partir de sua estrutura.

O esquema estrutural para a implementação do *Strategy* é apresentado no diagrama de classes da figura a seguir. Para implementar o padrão *Strategy*, uma interface fornece o acesso aos algoritmos enquanto classes concretas implementam essa interface, implementando cada alternativa de algoritmo. Seguindo o Princípio de Inversão de Dependência (D do SOLID, visto anteriormente), as classes clientes devem depender somente da interface. Dessa forma, a estratégia empregada por cada cliente pode ser alterada em tempo de execução, trocando a instância da classe concreta que está sendo acessada através da interface. O padrão pode também ser implementado trocando a interface por uma classe abstrata.

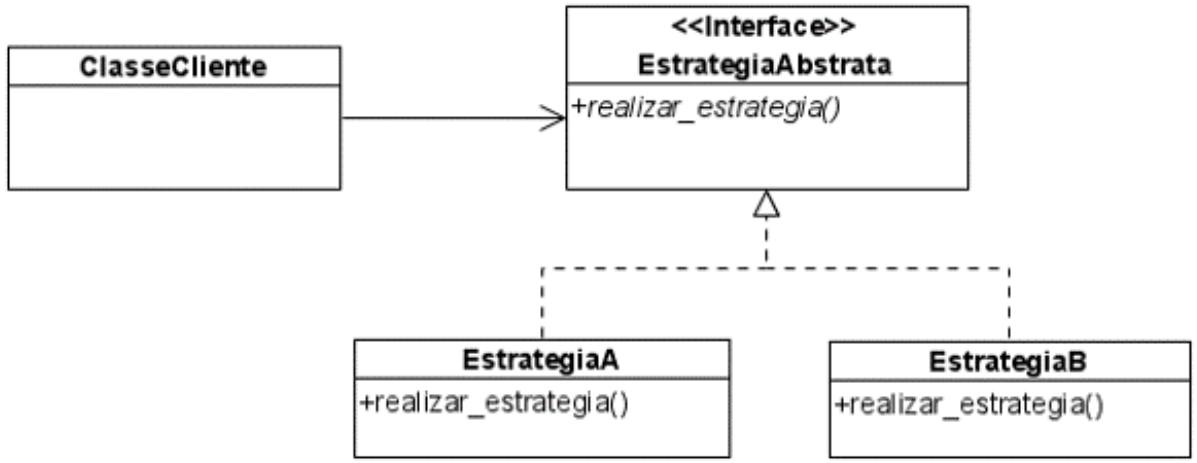


Figura 14.5: Esquema estrutural do padrão de projeto Strategy.

Vejamos um exemplo de aplicação do padrão *Strategy* em um sistema de *Machine Learning*. Vimos na seção anterior que uma boa prática é separar as diferentes responsabilidades do sistema em classes distintas (S do SOLID). Uma responsabilidade poderia ser a

de pré-processar os dados. Além disso, vimos na parte de *Machine Learning* que uma das tarefas que conceitualmente faz parte do pré-processamento é a seleção de atributos (*feature selection*), e que há diferentes formas de realizar essa tarefa, como a **seleção univariada**, **eliminação recursiva** e a **importância de atributos**.

Aplicando o padrão *Strategy*, conforme a figura a seguir, a nossa classe `PreProcessador` poderia trabalhar com diferentes estratégias de *feature selection*, que poderiam ser alteradas em tempo de execução ao invocar o método `setEstrategia`, passando uma instância da estratégia concreta a ser utilizada. Isso permitiria, por exemplo, reconfigurar a estratégia de *feature selection* do *pipeline* de *Machine Learning* no caso de resultados não satisfatórios, trocando simplesmente a estratégia do `PreProcessador` e repetindo a execução do *pipeline* sem alterar nada nas demais classes. Esse *pipeline* poderia automatizar todo o ciclo de MLOps, por exemplo, em que as ações de pré-processamento representariam apenas alguns de seus passos.

Note que, assim como para a *feature selection* do pré-processamento, o *pipeline* de *Machine Learning* como um todo tem diversos pontos em que diferentes estratégias podem ser utilizadas e em que o padrão de projeto *Strategy* poderia ser aplicado como uma solução elegante, incluindo diferentes algoritmos de aprendizado, diferentes formas de avaliação de modelos, entre muitos outros.

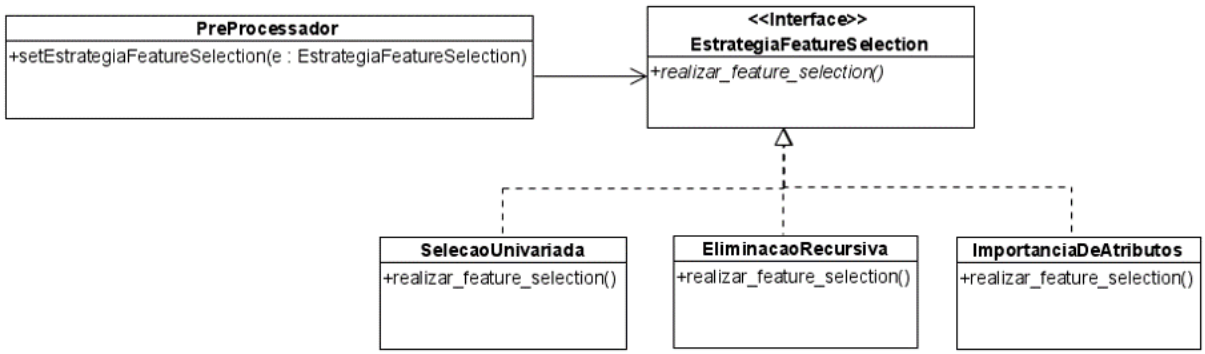


Figura 14.6: Aplicação do padrão de projeto Strategy para feature selection.

Na sequência, apresentamos mais alguns exemplos de padrões de projeto que podem trazer significativos benefícios para sistemas de *Machine Learning*. Começaremos pelo padrão *Pipeline*, que já foi indiretamente citado no exemplo do *Strategy*:

- Padrão ***Pipeline***: Esse padrão fornece a capacidade de construir e executar uma sequência de operações. Uma implementação (adaptada) desse padrão pode ser encontrada na própria biblioteca do Scikit-learn. O padrão *Pipeline* é usado para dividir um processo complexo em partes menores e mais gerenciáveis. Isso pode ser útil em sistemas de *Machine Learning*, onde as etapas incluem o pré-processamento, o treinamento de modelos e a avaliação dos seus resultados. Cada etapa pode ser encapsulada em um componente do *pipeline*, facilitando a modificação e o ajuste do processo.
- Padrão ***Observer***: Esse padrão também faz parte dos padrões originalmente propostos por Gamma *et al.* (1995). Ele permite notificar um conjunto de objetos quando ocorre uma mudança em outro objeto. Isso pode ser útil em sistemas de *Machine Learning*, onde os resultados dos modelos podem mudar à medida que novos dados são adicionados. A observação dessas mudanças poderia disparar ações para realizar ajustes no modelo. Em uma estratégia de MLOps automatizada, o padrão *Observer* fornece uma maneira elegante de implementar o esqueleto estrutural de um *drift monitor*, que monitora mudanças nos resultados das predições e notifica outros objetos que precisam tomar ações de verificação e/ou atualização sempre que isso acontece.
- Padrão **Injeção de Dependência**: Esse padrão é construído com base nos Princípios de *Inversão de Dependência* (DIP - o D do SOLID) e *Inversão de Controle* (FOWLER, 2004). Enquanto a inversão de dependência define que devemos depender de abstrações e não de implementações, a inversão de controle diz respeito a passar o controle da aplicação a um módulo específico, em que as abstrações são instanciadas. De forma resumida, esse padrão permite que as dependências de uma

classe sejam injetadas a partir de uma fonte externa, em vez de serem criadas dentro da classe (LAIGNER *et al.*, 2022). Isso pode ser útil em sistemas de *Machine Learning*, onde diferentes partes do sistema podem depender de diferentes instâncias de classes de bibliotecas ou *frameworks* de *Machine Learning*. O uso desse padrão pode tornar o sistema mais modular e fácil de manter, por exemplo, ao separar o código referente à aplicação dos algoritmos de *Machine Learning* do código de configuração das dependências, o que permite a fácil substituição dessas dependências quando necessário.

Esses quatro são apenas alguns exemplos de padrões de projeto que podem ser aplicados a sistemas de *Machine Learning*. O uso desses padrões, de forma ciente em relação às vantagens e desvantagens de sua aplicação, pode ajudar a tornar o sistema mais modular e adaptável, permitindo que os desenvolvedores criem sistemas de *Machine Learning* mais robustos e eficazes. Além disso, os padrões acabam estabelecendo um vocabulário comum entre desenvolvedores para comunicar ideias relacionadas ao projeto.

Enquanto, após um primeiro contato, os padrões de projeto possam parecer complexos, depois que você se acostuma com eles, seu uso passa a ser bastante intuitivo. Quase todos eles utilizam conceitos básicos da programação orientada a objetos: *herança*, *polimorfismo* e *composição*. Se você compreendeu a seção sobre os princípios SOLID, então você está pronto(a) para estudar, compreender e aplicar os padrões de projeto. Assim, uma sugestão é iniciar o estudo pelos princípios de projeto (o que pode ser feito neste livro) e depois seguir para os 23 padrões de projeto introduzidos por Gamma *et al.* (1995), que são clássicos, amplamente conhecidos por desenvolvedores de software do mercado e que fornecem uma boa base para compreender outros padrões contemporâneos.

Para finalizar esta seção, é importante destacar que estamos falando da aplicação de padrões de projeto de software para construir sistemas de *Machine Learning*. Existe uma literatura paralela sobre ***padrões de design de Machine Learning***, como a encontrada no

livro *Machine Learning Design Patterns*, de Lakshmanan *et al.* (2020). Estes não representam uma discussão de Engenharia de Software, mas soluções para desafios comuns encontrados durante a preparação dos dados, a construção de modelos e estratégias de MLOps. Ou seja, não focam em Engenharia de Software para Ciência de Dados e estão fora do escopo do nosso livro.

Idealmente, um bom ou uma boa cientista de dados conhecerá bem ambos os tipos de padrões: os padrões de projeto de software (que solucionam problemas relacionados com a engenharia do software) e os padrões de *Machine Learning* (que solucionam problemas relacionados com ciência de dados). Lembre-se de que, na prática, apenas uma pequena parte do código de um sistema de software inteligente é composta pelo código de *Machine Learning* (a que padrões de projeto também se aplicam, como vimos nesta seção). A infraestrutura de software requerida para o sistema como um todo tende a ser vasta e complexa (SCULLEY *et al.*, 2015) e deve ser pensada aplicando as melhores práticas de Engenharia de Software.

14.4 Code smells, dívida técnica e refatorações em sistemas de Machine Learning

Segundo Fowler (2018), um ***code smell*** é uma indicação superficial que geralmente corresponde a um problema mais profundo no sistema. Ou seja, um *code smell* é uma característica no código-fonte de um programa que possivelmente indica um problema estrutural mais profundo. A presença de *code smells* é um indicador da presença de *dívida técnica*.

Dívida técnica é um conceito introduzido por Ward Cunningham (1992) e representa a consequência de atalhos ou desconhecimento que levam à implementação de soluções não ideais. Um projeto de software que negligencia o foco na qualidade e na solução ideal (de forma intencional ou não intencional) tende a incorrer em custos

futuros: "juros", a serem reembolsados na forma de tempo adicional de desenvolvimento necessário a cada necessidade de manutenção, seja ela corretiva (para corrigir defeitos) ou evolutiva (para acrescentar novas funcionalidades). Pagar a dívida técnica rapidamente evita o acúmulo de "juros", daí a analogia com o conceito de dívida financeira. A dívida técnica representa um problema crítico para a indústria de software, com impacto de bilhões de dólares (TOM *et al.*, 2013).

No caso de sistemas de software inteligentes envolvendo *Machine Learning*, além do modelo, tais sistemas tipicamente envolvem subsistemas complexos, que compõem as atividades requeridas para um aprendizado de máquina eficaz e sua disponibilização para os usuários, o que envolve coletar, analisar e preparar os dados, treinar o modelo, validá-lo, colocá-lo em operação, monitorá-lo e retreiná-lo sempre que necessário.

No artigo *Machine Learning: The High-Interest Credit Card of Technical Debt* (SCULLEY *et al.*, 2014), os autores chamam a atenção para a dívida técnica nesse contexto e para a facilidade de gerar custos extremamente elevados de manutenção nesse tipo de sistema. Segundo Sculley *et al.* (2015), somente uma pequena fração de um sistema de *Machine Learning* do mundo real é composta pelo código de *Machine Learning* e a infraestrutura que circunda esse código é vasta e complexa. Esses sistemas estão sujeitos tanto à dívida técnica clássica quanto à dívida técnica especificamente relacionada a eles (TANG *et al.*, 2021).

Embora existam outros tipos de dívida técnica (RIOS *et al.*, 2018) que também se mostraram relevantes de acordo com pesquisas recentes, por exemplo, relacionadas com requisitos (MENDES *et al.*, 2016) e usabilidade (LAGE *et al.*, 2019), neste capítulo nosso foco é na dívida técnica referente ao projeto de software, e um bom indicador para sua presença são os *code smells*.

A lista a seguir, atualizada e documentada por Beck e Fowler (FOWLER, 2018), apresenta um resumo propositalmente sucinto dos

24 *code smells*:

1. **Nome misterioso**: ocorre quando métodos, módulos, variáveis ou classes são nomeados de uma forma que não comunicam o que fazem ou como usá-los. Esse *code smell* é frequente em código de *Machine Learning* (VAN OORT *et al.*, 2021). Entretanto, é preciso ter cautela na hora de interpretar essa informação, já que algumas convenções da área inspiradas na notação matemática (por exemplo, usar X para as entradas e Y para as saídas do modelo) acabam não sendo compatíveis com boas práticas da área de programação para nomes de variáveis (por exemplo, as recomendações do livro *Código Limpo* para nomear variáveis ou as convenções da PEP-8 para nomear variáveis em Python), mas ainda assim são intuitivas para quem atua na área.

2. **Código duplicado**: ocorre quando código idêntico ou muito semelhante existe em mais de um local. Infelizmente esse é um *code smell* ainda mais comum do que se gostaria em código relacionado com *Machine Learning* (VAN OORT *et al.*, 2021), indicando a necessidade do aprofundamento em boas práticas de projeto de software por parte de engenheiros(as) de *Machine Learning*.

3. **Método longo**: ocorre quando um método é muito grande e deveria ser quebrado em métodos menores que representem comportamentos mais inteligíveis.

4. **Muitos parâmetros**: ocorre quando um método tem uma longa lista de parâmetros, que é difícil de ler e torna a chamada e o teste do método complicados. Isso pode indicar que o propósito do método é mal-concebido e que o código deve ser refatorado para que a responsabilidade seja atribuída de maneira mais clara.

5. **Dados globais**: ocorre quando são utilizadas variáveis de escopo global, que devem ser evitadas. Elas podem levar a

dependências implícitas entre os códigos que manipulam essas variáveis e *dados mutáveis*.

6. **Dados mutáveis**: ocorre quando mudanças em dados podem levar a consequências indesejadas e defeitos difíceis de corrigir. Deve-se evitar alterar dados utilizados por diferentes partes do programa, por exemplo, alterando uma cópia para atender a necessidades específicas. Além disso, é recomendado proteger os dados com o devido encapsulamento.

7. **Mudança divergente**: ocorre quando um módulo é frequentemente alterado e por diferentes razões. Uma forma de evitar esse *code smell* é investir na coesão, aplicar o *Single Responsibility Principle* do SOLID e controlar o acoplamento.

8. ***Shotgun surgery***: ocorre quando uma única mudança precisa ser aplicada a várias classes ao mesmo tempo para ser realizada. De forma similar à mudança divergente, uma forma de evitar esse problema é investir na coesão e no controle do acoplamento. Os métodos que estão relacionados com uma mesma responsabilidade deveriam estar na mesma classe.

9. **Inveja de recurso** (*Feature Envy*): ocorre quando uma classe usa métodos de outra classe excessivamente. Uma forma de tratar esse problema pode ser mover os métodos para a classe que efetivamente mais os utiliza, o que pode impactar também na redução do acoplamento.

10. **Grupos de dados** (*Data Clumps*): ocorre quando um grupo de variáveis é frequentemente passado em conjunto por várias partes do programa. Provavelmente esses dados deveriam ter sido formalmente agrupados em um único objeto a ser passado entre as partes do programa.

11. **Obsessão com primitivos** (*Primitive Obsession*): ocorre quando se utiliza excessivamente os tipos primitivos da linguagem de programação (`int` , `float` etc.) em vez de criar

classes úteis para o domínio específico do problema em questão.

12. **Repetição de *switch*** (*Repeated Switches*): similar a um *if-tipado*, ocorre quando o código de uma classe testa o tipo frequentemente através de comandos *switch*. Isso deve ser evitado, principalmente quando ocorre diversas vezes nos diferentes métodos de uma classe. Nesse caso, deve-se considerar substituir o condicional fazendo uso de polimorfismo, criando classes para cada subtipo e evitando a necessidade do *if-tipado* ou do comando *switch*.

13. **Laços de repetição** (*Loops*): laços de repetição (`for` , `while` etc.) em geral podem ser substituídos por operações em *pipelines*. Operações sobre *pipelines*, como `filters` e `maps` , ajudam a facilmente ver os elementos envolvidos no processamento e o que está sendo feito com eles. Isso é particularmente relevante e muito utilizado no caso de códigos de *Machine Learning*, como vimos na parte deste livro referente a esse tópico.

14. **Elemento preguiçoso** (*Lazy Element*): ocorre quando um elemento (classe, método etc.) foi criado para dar estrutura ao programa, mas acaba fazendo muito pouco no sistema como um todo.

15. **Generalização especulativa**: ocorre quando se cria abstrações que acabam sendo pouco utilizadas, investindo esforços em uma generalização, por exemplo, pensando em possibilidades de reutilização futura, que acabam nunca sendo recompensadas.

16. **Atributos temporários**: ocorre quando uma classe possui atributos que só são utilizados em certas circunstâncias, levando o mantenedor à necessidade de entender essas circunstâncias, adicionando complexidade desnecessária. O ideal é abstrair esse conjunto de atributos para outra classe, levando também os métodos que manipulam esses atributos.

17. **Encadeamento de mensagens**: ocorre quando uma classe contém código que sucessivamente solicita outro objeto para, a partir deste, solicitar um novo objeto (umObjeto.getA().getB().getC().metodo()), para finalmente acionar o método. O encadeamento de mensagens gera dependências entre as classes envolvidas. Esse problema pode ser amenizado aplicando a *Lei de Demeter* (também conhecida como princípio 'não fale com estranhos'), escondendo as delegações e fazendo com que a comunicação flua somente entre objetos de classes que já estão semanticamente relacionadas, evitando gerar novas dependências.

18. **Intermediário** (*Middle Man*): ocorre quando muitos métodos da interface de uma classe estão delegando sua lógica para métodos de outra classe.

19. **Negociação privilegiada** (*Insider Trading*): ocorre quando há uma comunicação (reduzida e não bem definida) entre dois objetos que poderia ser evitada, por exemplo, ao mover dados ou funções ou ao estabelecer um intermediário mais coeso para mediar a comunicação que realmente for necessária.

20. **Classe grande**: ocorre quando uma classe fica grande e excessivamente complexa por conter muitos tipos ou muitos métodos não relacionados. Esse *code smell* também é conhecido como *God Class*. Ele é percebido como crítico por desenvolvedores durante atividades de manutenção (FERNANDES; KALINOWSKI, 2023).

21. **Classes alternativas com diferentes interfaces**: ocorre quando classes representam alternativas de solução, mas possuem interfaces incompatíveis. Nesse caso, deve-se abstrair uma interface comum para estas classes e aplicar o Princípio de Inversão de Dependência (o D do SOLID).

22. **Classes de dados**: ocorre quando se tem classes que possuem somente atributos e métodos *getters* e *setters* para obter e

alterar seus valores, sem nenhuma (ou pouca) lógica adicional. Nesse caso, deve-se inicialmente assegurar que os dados tenham coesão e verificar se o seu encapsulamento está adequado, para então buscar mover métodos que manipulem seus dados para o seu interior, sempre respeitando os princípios básicos da orientação a objetos que vimos no início deste capítulo.

23. **Legado recusado**: ocorre quando subclasses herdam métodos e dados mas acabam não fazendo uso (ou fazendo uso muito limitado) dos recursos herdados. Nesse caso, deve-se repensar quais métodos realmente deveriam fazer parte da classe mais genérica e eventualmente substituir a *herança* por *delegação*. A substituição da herança por delegação (ou o sentido inverso), além de considerar o uso efetivo dos recursos herdados, deve levar em consideração também se a subclasse em termos semânticos realmente representa um subtipo da classe mais genérica. Nos casos em que isso não ocorre, deve-se optar pela delegação.

24. **Comentários**: ocorre quando se tem comentários excessivos. Fowler (2018) sugere que antes de escrever comentários se busque refatorar o código de modo que o código fique intuitivo e o comentário se torne desnecessário. Não é preciso exagerar na remoção desse *code smell*. O próprio Fowler (2018) reconhece que comentários, quando usados com parcimônia, podem ser úteis para, por exemplo, explicar razões para implementar algo de determinada forma e fornecer outras informações que podem se mostrar úteis para apoiar a manutenção futura.

Existem diversos *code smells* adicionais documentados em outras fontes e capturados por diferentes tipos de ferramentas de análise estática (veja mais sobre análise estática no capítulo seguinte). Além dos 24 da lista de Fowler (2018), dois *code smells* bastante conhecidos são a *complexidade planejada* e a *complexidade ciclomática*.

A **complexidade planejada** envolve o uso forçado de padrões de projeto introduzindo uma complexidade desnecessária onde uma solução mais simples atenderia. Esse padrão ocorre quando se busca aplicar um padrão em situações em que o problema que o padrão trata não se manifesta (ou não se manifesta de maneira suficiente), ou sem considerar suas consequências. Já a **complexidade ciclomática** indica um número elevado de comandos de desvio de fluxo de controle (`if` / `else` , `switch` , `for` , `while` etc.) de um método, indicando que o método deveria ser dividido em funções menores ou que tem potencial para simplificação.

Investigações recentes indicam a ocorrência frequente de *code smells* em projetos de sistemas de software inteligentes envolvendo *Machine Learning*, além de outras deficiências que levam a dificuldades de manutenção e reprodutibilidade de projetos de *Machine Learning* (VAN OORT *et al*., 2021), em particular, referentes à gestão de configurações e dependências em projetos Python. No capítulo seguinte, falaremos um pouco mais sobre a reprodutibilidade de códigos de *Machine Learning*. Uma outra fonte interessante sobre formas como sistemas de software inteligentes envolvendo *Machine Learning* podem ser particularmente propensos a gerar dívida técnica é o artigo de Sculley *et al.* (2015).

Refatoração é o processo de alteração de um sistema de software de modo que o comportamento externo do código não mude, mas que sua estrutura interna seja melhorada (FOWLER, 2018). É uma maneira disciplinada de aperfeiçoar o código que facilita a manutenção futura e minimiza a chance de introdução de defeitos. Quando você usa refatoração, você está melhorando o projeto do código após a escrita. A refatoração frequente, por exemplo, com a remoção de *code smells*, é uma maneira de reduzir a dívida técnica contida no projeto de um sistema de software.

Um catálogo completo de refatorações pode ser obtido no livro de Fowler (2018). Em sua essência, essas refatorações envolvem a reestruturação de código fonte para refletir adequadamente os princípios de projeto apresentados ao longo deste capítulo. Ou seja,

embora a leitura do catálogo das refatorações possa ser didaticamente útil, ao refletir sobre como alterar o código buscando a correta aplicação dos princípios de projeto, acabamos aplicando a refatoração implicitamente.

No contexto de sistemas de software inteligentes envolvendo *Machine Learning*, achados de uma pesquisa recente indicam que, embora o código de *Machine Learning* tenda a corresponder à menor parte do sistema, ele tende a ser o mais frequentemente refatorado (TANG *et al.*, 2021). De forma consistente com o estudo anteriormente citado sobre a prevalência de *code smells* em código de *Machine Learning*, essa pesquisa revelou que as refatorações mais frequentes tratavam da reorganização de códigos para a melhoria de performance ou para a remoção de *code smells* como código duplicado, grupos de dados e obsessão com primitivos, entre outros. Em muitos casos, a reorganização envolveu a aplicação de técnicas já conhecidas de refatoração a aspectos específicos de *Machine Learning*, como introduzir uma herança para tratar *if-tipados* com polimorfismo para a aplicação de diferentes tipos de modelos de classificação, subir aspectos comuns a mais de um modelo na hierarquia e descer hiperparâmetros de uma classe mais abstrata para subclasses específicas.

Reforçamos a importância da refatoração frequente de sistemas de software inteligentes para evitar dívida técnica referente tanto ao código de *Machine Learning* quanto ao código de infraestrutura, tendo em vista que a dívida técnica costuma ter implicações severas e juros altíssimos em projetos de *Machine Learning*. Considerando essas características e as implicações financeiras diretas confirmadas por evidências científicas, a tendência é que as empresas passem a valorizar cada vez mais engenheiros(as) de *Machine Learning* que também conheçam bem de Engenharia de Software.

CAPÍTULO 15
Controle da qualidade de sistemas de software inteligentes

O controle da qualidade no contexto de sistemas de software inteligentes envolve preocupações específicas em relação à qualidade dos componentes de *Machine Learning*. Os requisitos especificados para o componente de *Machine Learning*, utilizando técnicas como PerSpecML, precisam ser verificados.

Estes requisitos podem dizer respeito a:

- **Qualidade dos dados** (por exemplo, acurácia, consistência, completude, ausência de viés, entre outros);
- **Aspectos do modelo** (por exemplo, o tamanho do modelo, o tempo de treinamento, o tempo de inferência, a confiabilidade do modelo medida através de métricas específicas, a degradação de performance, a explicabilidade das inferências, entre outros);
- **Aspectos de infraestrutura**, como custo, telemetria, monitorabilidade, capacidade de aprendizado incremental e aspectos de MLOps, entre outros;
- **Aspectos de experiência do usuário**, como requisitos relacionados à visualização da informação, interatividade, valor agregado ao usuário, entre outros;
- **Aspectos relacionados com o alcance dos objetivos** de maneira geral, como o atendimento a requisitos relacionados ao *trade-off* do uso do modelo, atendimento de *leading indicators*, verificação do alcance dos objetivos do cliente e organizacionais através da avaliação de hipóteses de negócio, entre outros.

Não há bala de prata para realizar esse controle da qualidade, e a estratégia de controle da qualidade dependerá fortemente dos requisitos e de conhecimento de Engenharia de Software a respeito de como estes requisitos podem ser verificados. Uma fonte que pode

ser consultada como ponto de partida a respeito de quais métodos de verificação e validação empregar para verificar quais características de qualidade pode ser encontrada em Mendoza *et al.* (2019), que relaciona 19 métodos de verificação e validação com as 8 características de qualidade da ISO 25010. Entretanto, nesta pesquisa foram considerados essencialmente sistemas convencionais e as 8 características de qualidade da ISO 25010 se desdobram fortemente em contextos de *Machine Learning*, como nos diversos exemplos de possíveis necessidades de verificação apresentados anteriormente.

Neste capítulo, de forma não exaustiva, faremos recomendações práticas que consideramos relevantes para o controle da qualidade de sistemas de software inteligentes, em que modelos são comumente construídos utilizando programação literária interativa em *notebooks* e, depois de prontos, integrados com o restante do sistema, através de alternativas de arquiteturas como as anteriormente vistas. As nossas considerações são baseadas tanto na literatura quanto em experiências práticas obtidas em projetos de Ciência de Dados e abordam recomendações referentes à reprodutibilidade de *notebooks* e a como técnicas clássicas de controle da qualidade, como análise estática, revisões de código modernas e o teste de software, podem ser aplicadas a sistemas de software inteligentes.

15.1 Reprodutibilidade de notebooks

Como vimos, na área de Ciência de Dados é extremamente comum realizar tarefas como análise exploratória, pré-processamento e avaliação de algoritmos de *Machine Learning* utilizando programação literária interativa (*notebooks*). Entretanto, a forma interativa de produção do código pode prejudicar sua reprodutibilidade. Em um estudo que analisou *notebooks* disponíveis no GitHub, foi identificado que somente 25% dos *notebooks* permitem a execução

de todas as suas células e que apenas 10% produzem resultados próximos aos armazenados nos *notebooks* (PIMENTEL *et al.*, 2021).

A partir desse estudo, foram compiladas algumas boas práticas visando à reprodutibilidade dos *notebooks*, que compartilhamos a seguir:

- Usar títulos descritivos com um conjunto restrito de caracteres (A-Z, a-z, 0-9) e células *markdown* com títulos mais detalhados no corpo;
- Prestar atenção ao final do *notebook* e verificar se as células do final foram executadas e se não cabe nenhuma célula de *markdown* para concluir a análise;
- Abstrair funções, classes e módulos que possam ser importados por diversos *notebooks* e testados externamente;
- Declarar dependências em arquivos apropriados (*requirements.txt*) e fixar a versão dos módulos;
- Usar um ambiente limpo para testar dependências e verificar se todas estão declaradas corretamente;
- Colocar importações no início do *notebook*;
- Usar caminhos relativos para acessar dados no repositório;
- Reexecutar *notebooks* do início ao fim antes de enviá-los para algum repositório.

Essas são práticas intuitivas, mas que podem se mostrar relevantes para melhorar a reprodutibilidade de *notebooks*, o que pode se mostrar essencial principalmente em momentos em que modelos construídos utilizando *notebooks* precisam ser atualizados.

15.2 Análise estática

Análise estática é o processo de analisar automaticamente um código com base na sua estrutura, sem envolver sua execução, ou seja, de forma estática (WAGNER, 2013). Ferramentas de análise

estática são comumente chamadas de *Linters*. Essas ferramentas em geral são capazes de verificar se guias de estilo estão sendo seguidos, encontrar determinados padrões de defeitos e analisar aspectos de fluxo de controle e fluxo de dados dos programas. Entretanto, um analisador estático não será capaz de dizer se o programa faz o que ele se propõe a fazer – ou seja, a análise estática, embora útil, precisa ser complementada com outras formas de controle da qualidade.

Os *Linters* mais usados no mercado no contexto do Python são o **Flake8** e o **Pylint**. Ambos verificam o guia de estilos PEP-8, padrões de defeitos e aspectos de fluxo de controle e fluxo de dados dos programas, além de outras opções. Embora haja alguma sobreposição, não há problema em combinar ferramentas de análise estática. Tanto o Flake8 quanto o Pylint podem ser utilizados com *notebooks* e é possível encontrar adaptações para seu uso no contexto específico de *notebooks* – por exemplo, o ***flake8-nb***, que é uma ferramenta que pode ser executada via linha de comando para realizar análise estática de *notebooks*.

Vejamos um exemplo de utilização do *flake8-nb*. Em um *notebook* denominado *notebook_exemplo.ipynb*, imagine que a 20ª célula tenha o seguinte código, com um espaço faltando após o : na definição do dicionário:

```
dict_mal_formatado = {"nome":"Kalinowski"}
```

A execução do *flake8-nb* resultaria na seguinte saída:

```
$ flake8_nb notebook_exemplo.ipynb
notebook_exemplo.ipynb#In[20]:1:28: E231 missing whitespace after ':'
```

É possível observar que o *flake8_nb* detectou o problema de formatação na célula 20 (linha 1, coluna 28). Para mais informações sobre o *flake8-nb*, sugerimos recorrer à documentação oficial, que se encontra disponível em https://flake8-nb.readthedocs.io/en/latest/index.html.

Além do *flake8-nb*, que essencialmente estende os conceitos de análise estática do Flake8 para considerar as células de um *notebook*, recentemente foram propostos no meio científico *linters* específicos para aspectos de Ciência de Dados, elaborados para considerar, por exemplo, as recomendações de reprodutibilidade de *notebooks* da seção anterior. Exemplos dessas ferramentas são as desenvolvidas no contexto de duas recentes pesquisas de doutorado no Brasil (PIMENTEL *et al.*, 2021) e na Itália (QUARANTA *et al.*, 2022). Entretanto, essas ferramentas ainda não alcançaram o estado da prática e, por isso, mesmo sugerindo que as experimentem para avaliar sua adequação para os seus projetos específicos, estamos evitando recomendações mais enfáticas neste momento.

15.3 Revisões de código modernas

Revisões de código modernas desempenham um importante papel no controle da qualidade de software. Elas permitem que defeitos que o desenvolvedor deixou passar durante o desenvolvimento sejam corrigidos antes que passem para o repositório de código compartilhado. Essencialmente, revisores refletem sobre a qualidade do código de um desenvolvedor, referente a pequenas mudanças no sistema, antes que essas mudanças sejam integradas no repositório de código.

Os defeitos encontrados são passados para que a pessoa autora possa corrigir e submeter uma nova versão corrigida. O que caracteriza as revisões de código como "modernas" (embora o termo tenha sido definido há mais de 10 anos) é a sua informalidade e sua integração leve no processo de desenvolvimento, permitindo obter *feedback* rápido sobre mudanças propostas no código.

A figura a seguir ilustra como o processo de revisão de código moderna ocorre no desenvolvimento. Neste exemplo, primeiro o autor realiza alterações no código e solicita a integração no

repositório (*Pull Request*). Os revisores então analisam a solicitação revisando as alterações e fornecendo comentários. O autor recebe o *feedback* dos revisores e submete uma versão melhorada. Na versão melhorada, o *Pull Request* é então aceito e as alterações são integradas no repositório.

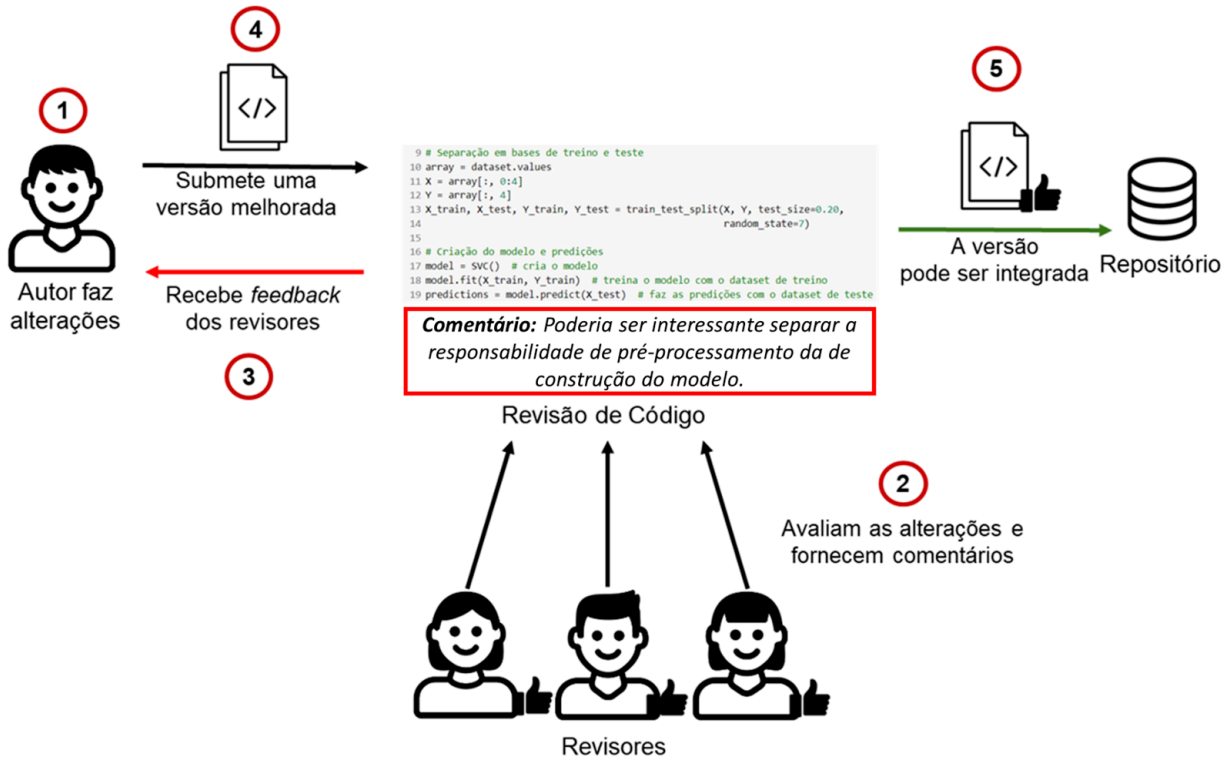


Figura 15.1: Exemplo do processo de revisão de código moderna no desenvolvimento.

Diversas empresas, como Google (SADOWSKI *et al.*, 2018) e Microsoft (BACCHELLI; BIRD, 2013) têm reportado benefícios da aplicação de revisões de código modernas e a prática tem se disseminado como uma das boas práticas da Engenharia de Software. De fato, elas permitem identificar não somente defeitos, como também vulnerabilidades de segurança e problemas estruturais no código. Além disso, tendem a tornar o código mais simples de ser mantido e promovem a transferência de conhecimento na equipe de desenvolvimento.

Naturalmente essa prática de controle da qualidade pode também ser aplicada no desenvolvimento de sistemas de software inteligentes envolvendo *Machine Learning*, já que a natureza do código sendo construído não afeta diretamente a prática. Basta utilizar apropriadamente ferramentas capazes de lidar com solicitações de mudança (como o Jira e o Microsoft Azure DevOps) e controle de versões (como o Git) para o código sendo construído. Normalmente o ambiente para viabilizar as revisões de código modernas é preparado pelo(a) analista de infraestrutura e DevOps.

15.4 Teste de sistemas de software inteligentes

Conceitos básicos

O teste de software é realizado em diferentes fases ao longo do desenvolvimento: **teste unitário**, **teste de integração**, **teste de sistema** e **teste de aceitação**. O *teste unitário* se refere à fase do processo de teste em que se testam as menores unidades de software desenvolvidas. O *teste de integração* envolve, a partir dos módulos testados no nível de unidade, testar a estrutura de programa que foi determinada pelo projeto. O *teste de sistema* envolve executar o sistema sob o ponto de vista de seu usuário final, varrendo as funcionalidades em busca de falhas. Por fim, o *teste de aceitação*, eventualmente também chamado de homologação, é realizado pelo cliente. Além disso, como vimos, no contexto de sistemas de *Machine Learning*, o uso de **experimentação contínua** (teste A/B) em produção para testar hipóteses de negócio é bastante comum.

A figura a seguir destaca as perspectivas assumidas nas diferentes fases de teste e na experimentação contínua. Daremos

recomendações práticas para cada uma dessas fases mais adiante, com ênfase em aspectos relacionados com *Machine Learning*.

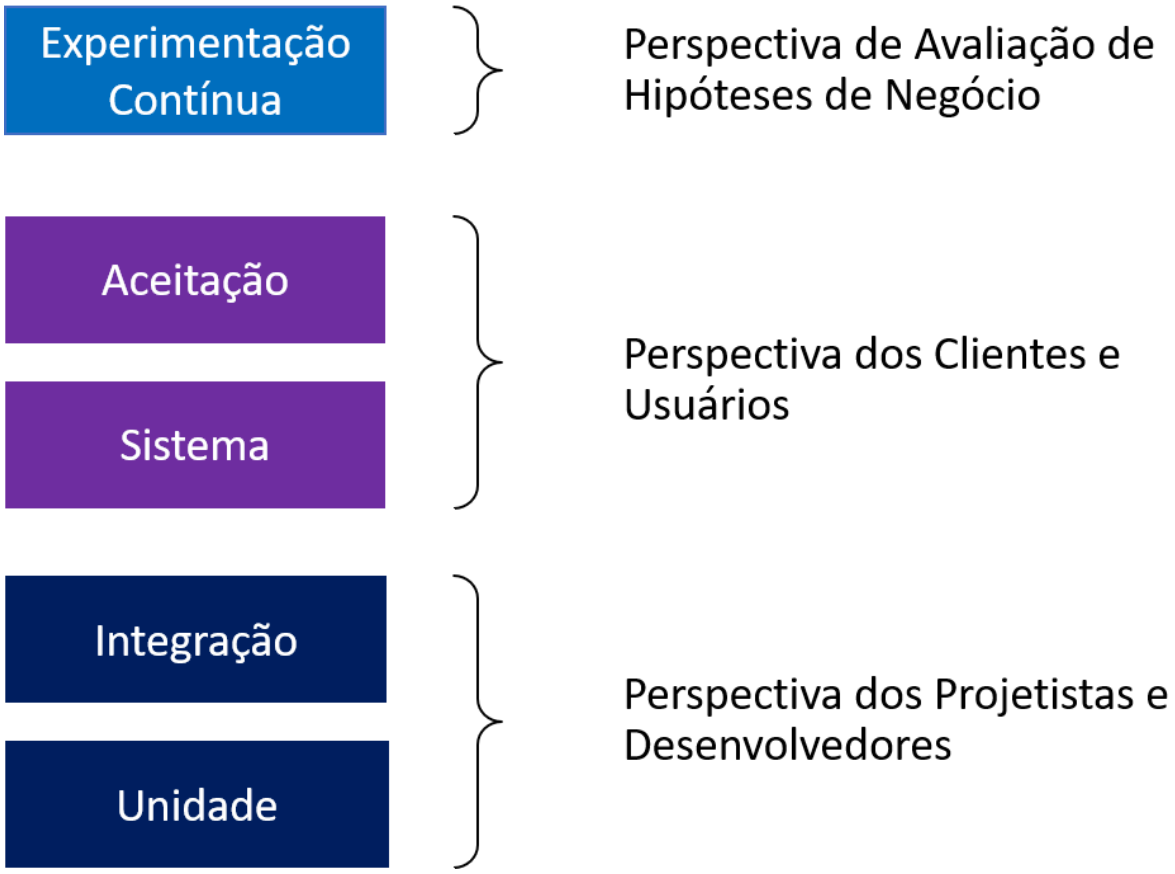


Figura 15.2: Fases de teste, experimentação contínua e perspectivas assumidas.

Antes de iniciar a discussão, é importante ressaltar que teste de software é uma disciplina ampla e que tem material muito rico disponível na literatura tradicional de Engenharia de Software, além do escopo deste livro. Para uma visão geral sobre teste de software ou para melhor compreender os conceitos apresentados nesta seção, recomendamos consultar o livro *Introdução ao Teste de Software*, de Delamaro *et al.* (2016). Já para uma abordagem mais focada na construção de testes eficazes pelo desenvolvedor, um excelente livro é o *Effective Software Testing: A Developer's Guide*, de Aniche (2022). Ressaltamos que estes livros não abordam diretamente o teste de sistemas de software inteligentes envolvendo

componentes de *Machine Learning* para os quais acrescentamos considerações específicas a seguir.

Considerações sobre o estado da arte

O teste de componentes de *Machine Learning* traz diversos desafios específicos e pesquisadores já começaram a adaptar conceitos e técnicas para este domínio. Um bom apanhado sobre pesquisas na área pode ser encontrado no artigo *On Testing Machine Learning Programs*, de Braiek e Khomh (2020). Para prover uma visão geral, na figura a seguir organizamos os tipos de técnicas de teste encontradas por Braiek e Khomh (2020) nas categorias apresentadas por Delamaro *et al.* (2016): *técnicas funcionais* (ou caixa preta), *técnicas estruturais* (ou caixa branca) e *técnicas baseadas em erros*. Explicaremos do que se tratam estas técnicas na sequência.

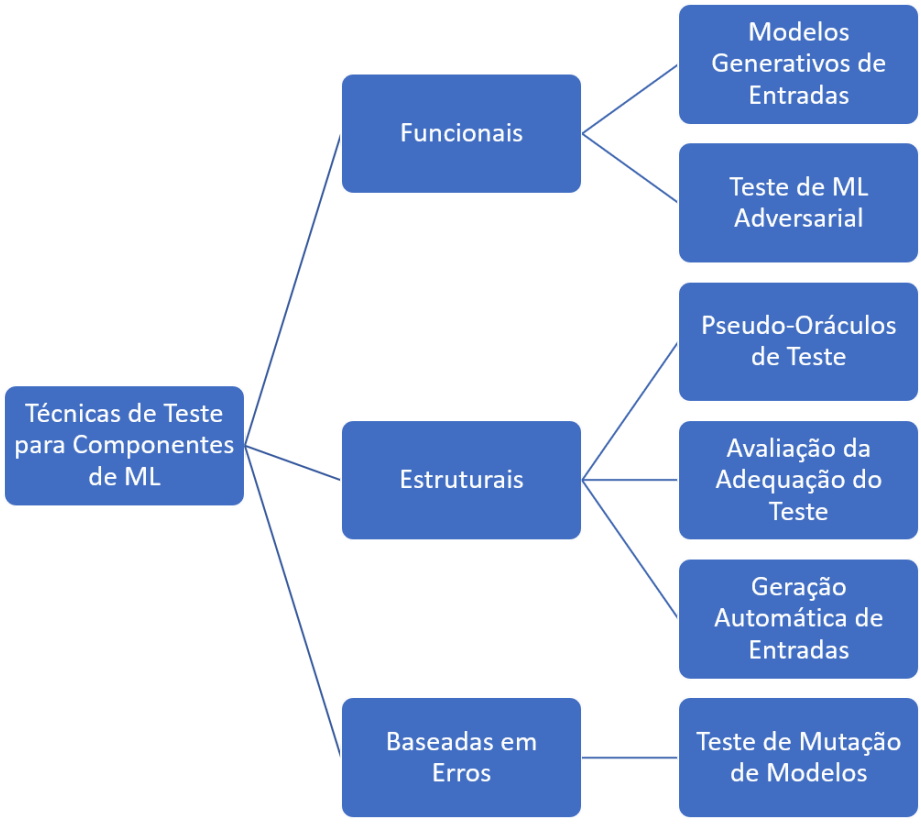


Figura 15.3: Técnicas de teste propostas para componentes de Machine Learning.

As **técnicas funcionais** avaliam o comportamento externo do componente (modelo) de *Machine Learning*, sem considerar o comportamento interno dele. Ou seja, elas avaliam o modelo em relação aos resultados esperados sem considerar detalhes da sua estrutura interna. As técnicas funcionais incluem:

- **Modelos generativos de entradas**, que consideram a distribuição de probabilidade de cada entrada para gerar quantos pontos de dados forem necessários para testar o modelo de *Machine Learning* em teste.
- **Teste de aprendizado de máquina adversarial**, que envolve a alteração de entradas para testar a robustez do modelo em relação a manipulações não triviais e potencialmente perigosas dos dados de entrada. Como o nome indica, no teste adversarial o modelo é tratado como um adversário e se busca contraexemplos para os quais o modelo possivelmente não dará a resposta esperada.

As **técnicas estruturais** envolvem elaborar casos de teste considerando o comportamento interno do componente (modelo) de *Machine Learning*. As técnicas estruturais incluem:

- **Pseudo-oráculos de teste**, que permitem identificar testes que falharam, distinguindo os comportamentos corretos e incorretos do modelo em teste quando comparado a um pseudo-oráculo. As estratégias de geração de pseudo-oráculos de teste exploradas em pesquisas incluem o teste diferencial (que utiliza diversos modelos para resolver o mesmo problema, fazendo com que eles sirvam como oráculos de referência cruzada) e o teste metamórfico (que permite identificar comportamentos errados pela violação de relações metamórficas de entradas que, de acordo com a lógica de inferência, deveriam resultar na mesma saída).
- **Avaliação da adequação do teste**, que envolve aplicar critérios de adequação para estimar a capacidade de revelar falhas de um determinado conjunto de testes, visando medir o grau/confiança de que o conjunto de testes é adequado. É

importante ressaltar que, no caso de componentes de *Machine Learning*, esses critérios envolvem elementos estruturais diferentes dos usuais, que, em geral, são relacionados com linhas de código e decisões (fluxo de controle) ou inicialização e uso de variáveis (fluxo de dados). Exemplos de critérios para componentes de *Machine Learning* incluem o critério de cobertura de neurônios em redes neurais artificiais introduzido por Pei *et al.* (2017), que estima a quantidade de lógica da rede neural explorada por um conjunto de entradas, e as variações deste critério, como os criados por Tian *et al.* (2018) e Guo *et al.* (2018), ou, ainda, adaptações complementares de critérios existentes, como a adaptação do MC/DC (*Modified Condition/Decision Coverage*) para *Machine Learning* proposta por Sun *et al.* (2018).
- **Geração automática de entradas**, que buscam explorar o espaço de entrada, através de diversos algoritmos de busca, geração e otimização de entradas, para produzir automaticamente casos de teste que sejam eficazes, em geral, por meio da maximização de um ou vários critérios de adequação de teste adotados (por exemplo, cobertura de neurônios de uma rede neural).

Por fim, as **técnicas baseadas em erros** envolvem derivar casos de teste a partir de erros frequentes cometidos na elaboração de componentes (modelos) de *Machine Learning*. Os casos de teste são avaliados com base em modelos que possuem defeitos/problemas de construção que refletem esses erros frequentes, para determinar se eles são capazes de revelar falhas referentes a esses defeitos/problemas. Nesta categoria, temos:

- **Teste de mutação de modelos**, que são técnicas similares ao teste de mutação de software convencional e envolvem a alteração de aspectos internos do modelo através da aplicação de operadores de mutação específicos para então avaliar se os casos de teste são capazes de revelar essas mutações. Ou seja, são uma abordagem para testar e melhorar os casos de testes.

> Um bom conjunto de casos de teste (por exemplo, incluindo dados representativos e também bons exemplos adversariais) deveria ser capaz de revelar falhas referentes aos defeitos/problemas presentes no modelo mutante. Assim, a análise dos defeitos/problemas que não revelaram falhas leva à melhoria do conjunto de casos de teste, incluindo novos casos que sejam capazes de revelar estas falhas, quando possível.

Convidamos os interessados e interessadas a se aprofundarem nas pesquisas ativas na área através do acesso direto aos artigos científicos. Os consolidados por Braiek e Khomh (2020) e uma busca por suas citações atuais pode servir como um bom ponto de partida. Apesar de a literatura sobre o tema ser vasta, com diversas publicações recentes nos principais veículos da área de Engenharia de Software, estes conceitos ainda são pouco utilizados na prática pelas empresas. Tendo isso em vista, apresentaremos a seguir algumas recomendações de cunho mais prático com base em lições aprendidas em nossos projetos, que podem ser aplicadas com mais facilidade em problemas da indústria.

Recomendações práticas

Em relação a teste de componentes de *Machine Learning*, independente da fase de teste, para testar as predições dos modelos, é preciso escolher métricas adequadas e *thresholds* (valores limite) aceitáveis para o desempenho do modelo. Isso pode ser feito em função dos requisitos definidos, para determinar se o teste passou ou falhou. Requisitos para um modelo de classificação poderiam envolver, por exemplo, uma acurácia acima de 85%. Já para um modelo de regressão poderiam envolver, por exemplo, um R2 *score* acima de 0,7.

A métrica e o *threshold* específico devem ser estabelecidos considerando os requisitos definidos. Por exemplo, uma acurácia de 95% para um sistema que classifica se uma pessoa tem COVID-19 ou não com base em outras informações (como a temperatura corporal, os sintomas apresentados, a quantidade de dias passados

desde o início dos sintomas, entre outros) deixaria muito a desejar. Até um classificador *dummy* que sempre retorne "não" como resposta será melhor do que este modelo, já que (no momento da escrita deste livro) mais de 99,5% das pessoas não estão com COVID-19.

Nesse caso, seria mais interessante olhar, por exemplo, para a capacidade de identificar quem tem a doença (alto *recall* da classe COVID-19) sem indicar junto diversas pessoas que não possuem a doença (alta *precision*). Ou seja, um sistema que atenda a um *recall* de 90% (identificando 90% das pessoas que tem a doença) com uma *precision* de 50% (identificando junto a estas uma parcela igual que não possui a doença) poderia se mostrar muito mais útil do que um com acurácia geral de 95%. A escolha das métricas e os *thresholds* são uma preocupação que deve ser tratada no início do projeto (BERRY, 2022; VILLAMIZAR *et al.*, 2022a).

Além de testar os requisitos de desempenho do modelo, dependendo dos requisitos levantados, outras características referentes ao modelo podem precisar ser verificadas, incluindo o tamanho do modelo, o tempo de treinamento, o tempo de inferência, entre outros. Ainda em relação ao componente de *Machine Learning*, além das características relacionadas à perspectiva do modelo, não podemos esquecer de testar as características relevantes das perspectivas de objetivos, experiência de usuário, infraestrutura e dados (VILLAMIZAR *et al.*, 2023).

Em relação aos **testes unitários**, idealmente as classes são testadas externamente aos *notebooks*. O **PyTest** é o principal *framework* para teste unitário em Python. É importante ressaltar que os testes de código orientado a objetos serão mais eficientes se forem construídos utilizando técnicas de teste estruturais, incluindo técnicas de fluxo de controle e de fluxo de dados da Engenharia de Software tradicional. Entretanto, como vimos, esses testes não são suficientes no contexto de sistemas de *Machine Learning*, em que o algoritmo pode estar correto, mas ainda assim o modelo ter

aprendido comportamentos errados a partir dos dados a que foi submetido.

Assim, na prática, é preciso monitorar de perto a qualidade dos dados. Para testar a predição dos modelos, é comum que se utilize um conjunto de dados validados pelo especialista de domínio em relação à sua qualidade (representatividade, corretude, consistência, credibilidade, etc.), comumente conhecido como *golden dataset*. Vale lembrar que é importante testar o modelo em dados não utilizados no treinamento para ver como ele se comporta em situações reais, avaliando a sua capacidade de generalização.

Além disso, o **teste de robustez**, que avalia o modelo com dados de entrada imprecisos ou incompletos, pode ajudar a identificar possíveis vulnerabilidades ou pontos fracos no modelo. A avaliação da qualidade dos dados e a detecção de dados de qualidade suspeita pode se beneficiar do conhecimento recente gerado a respeito de *data smells*, que indicam possíveis problemas nos dados (FOIDL *et al.*, 2022). Para mais detalhes sobre características de qualidade dos dados, recorra ao capítulo de requisitos de sistemas de software inteligentes.

Certamente o menor desafio nos testes unitários de código de *Machine Learning* será aprender a utilizar a sintaxe do *framework* para a escrita dos testes. O PyTest, por exemplo, é o *framework* para teste unitário mais utilizado para programas escritos na linguagem Python. Ele é bastante prático e simples de utilizar e identifica automaticamente os arquivos nomeados como `test_*.py` ou `*_test.py` e dentro deles as funções iniciadas com `test_` , e então as executa, reportando os resultados em função da avaliação de comandos `assert` . Instrumentar o seu código com testes unitários é uma boa prática, pois permitirá identificar o quanto antes se alguma ação realizada fará com que o sistema (ou o modelo) não atenda mais ao que se espera dele.

Vejamos um exemplo dessa notação. O código a seguir, contido em um arquivo fictício `test_model.py` , tem uma função de teste chamada

`test_model_prediction`, que verifica através de uma assertiva se a acurácia do modelo construído está superior a 85%. Cabe ressaltar que, para fins de simplificação e para focar o exemplo no teste, reutilizamos os códigos das classes `Carregador`, `PreProcessador`, `Modelo` e `Avaliador` do capítulo sobre projeto de sistemas de software inteligentes.

De forma resumida, o teste realiza a carga dos dados, o pré-processamento e o treinamento do modelo e depois verifica se a predição do modelo alcançou acurácia acima do *threshold* de 85%. É possível observar que os dados de teste (`X_test` e `Y_test`, referentes aos dados de teste de entrada e de saída) deste exemplo não foram utilizados durante o treinamento do modelo. Neste exemplo, como se trata de um *dataset* muito simples (o Iris, que já utilizamos anteriormente), o treinamento de um modelo SVM simples já alcança 86,6% de acurácia, o que é acima do *threshold* definido no teste, e o teste passa.

```
# from carregador import Carregador
# from preprocessador import PreProcessador
# from modelo import Modelo
# from avaliador import Avaliador

def test_model_prediction():
    url_dados = ('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data')
    atributos = ['comprimento_sepala', 'largura_sepala',
'comprimento_petala', 'largura_petala', 'especie']
    percentual_teste = 0.2

    # Instanciação das Classes
    carregador = Carregador()
    pre_processador = PreProcessador()
    modelo = Modelo()
    avaliador = Avaliador()

    # Carga
    dataset = carregador.carregar_dados(url_dados, atributos)
```

```
    # Pré-processamento
    X_train, X_test, Y_train, Y_test =
pre_processador.pre_processar(dataset, percentual_teste)
    # Treinamento do modelo
    model = modelo.treinar_SVM(X_train, Y_train)
    # Teste do resultado da avaliação com threshold de 85%
    assert avaliador.avaliar_acuracia(model, X_test, Y_test) > 0.85
```

Para fins didáticos, nesse exemplo (reutilizando as classes do capítulo de projeto de sistemas inteligentes) realizamos toda a carga, pré-processamento e treinamento para só depois testar a predição. Na prática, quando já se dispõe de um modelo, é possível somente carregar (ou acessar) esse modelo e invocar sua predição sobre os dados de teste (por exemplo, obtidos a partir de um *golden dataset* validado pelo especialista do domínio) para então comparar os resultados obtidos utilizando a métrica selecionada e o *threshold* predefinido. Informações práticas e exemplos sobre como carregar modelos a partir de arquivos ou sobre como acessar modelos disponibilizados através de um *endpoint* na nuvem podem ser encontradas no capítulo sobre implantação de modelos.

O **teste de integração** permitirá avaliar se as unidades sendo desenvolvidas pelos diferentes desenvolvedores funcionam em conjunto como parte do produto contido no repositório integrado. No caso de sistemas de software inteligentes, isso envolve testar a integração do modelo de *Machine Learning* com o restante do sistema. Esses testes ajudam a garantir que os modelos de *Machine Learning* estejam funcionando corretamente em conjunto com outras partes do sistema, como banco de dados, servidores, interfaces de usuário e outros componentes. É comum que para este tipo de teste o mesmo *golden dataset* seja utilizado. Testes de integração tendem a se beneficiar de práticas de integração contínua e automatização, como a geração automática de *builds* e testes automatizados.

O **teste de sistema** envolve executar o sistema sob o ponto de vista de seu usuário final, verificando tanto requisitos funcionais

quanto requisitos não funcionais do sistema como um todo (como segurança, desempenho, usabilidade, manutenibilidade, entre outros). Em relação aos testes de segurança, eles visam assegurar que os modelos de *Machine Learning* não apresentem vulnerabilidades significativas a ataques maliciosos.

Testes de desempenho, por sua vez, podem envolver medir o tempo de uma inferência ou a quantidade de inferências por segundo, para modelos que realizam processamento off-line em lote. Além disso, é importante testar a usabilidade do modelo através da interface do usuário. Neste contexto, a integração de práticas que beneficiem a experiência do usuário no processo de desenvolvimento deve ser considerada (FERREIRA *et al.*, 2023).

Já em relação à manutenibilidade, é importante testar requisitos referentes às expectativas de retreinamento do modelo. Como os dados mudam ao longo do tempo, é importante assegurar que o modelo possa ser mantido atualizado, o que em casos de soluções com estratégias completamente automatizadas de MLOps é bastante facilitado. Nesta fase, é importante ainda que se trabalhe com dados mais próximos aos dados reais de produção, evitando surpresas nos testes de aceitação.

O **teste de aceitação** pode ser realizado no ambiente de desenvolvimento simulando da melhor forma possível o ambiente de produção (teste *alpha*) ou, idealmente, no ambiente final de produção (teste *beta*). Eles são realizados pelo cliente e, independente de o teste ser *alpha* ou *beta*, neste momento estamos falando de testes com dados de produção.

No contexto de sistemas de *Machine Learning*, o uso de **experimentação contínua** (teste A/B) em produção é bastante comum. Nesse tipo de experimentação, duas alternativas de solução em produção com diferentes amostras de usuários são comparadas para avaliar (estatisticamente, com testes de hipótese, se possível) se uma solução inovadora está de fato trazendo ganhos (relacionados com hipóteses de negócio) quando comparada à

solução atualmente em vigor. Quando se trata de experimentar uma solução de inteligência computacional substituindo outra, é comum que se faça um *shadow deployment* (implantação simultânea da nova solução), para que ambas as soluções possam executar simultaneamente e os resultados possam ser comparados estatisticamente, permitindo chegar a uma conclusão sobre se a nova solução realmente traz os melhores resultados para o negócio.

CAPÍTULO 16
Gerência de configuração, DevOps e MLOps em sistemas inteligentes

16.1 Conceitos de gerência de configuração

Gerência de configuração é o processo de identificar, organizar e controlar modificações ao software sendo construído. Um projeto de desenvolvimento de software produz diversos itens, como o programa, documentação e dados. Esses conjuntos de itens são chamados, coletivamente, de configuração do software.

Para uma boa gerência de configuração, é necessário estar familiarizado com alguns conceitos. O primeiro deles é o conceito de ***baseline***, que se refere a uma configuração do software, em um ponto discreto do tempo, que foi formalmente revisada e aceita. A *baseline* serve como base para os passos posteriores do desenvolvimento.

No contexto ágil, é comum que se estabeleçam *baselines* após cada uma das *sprints* de desenvolvimento, contendo as novas funcionalidades entregues e aprovadas naquela *sprint*. Na prática, as *baselines* são estabelecidas utilizando o recurso de *tags* de sistemas de controle de versões como o Git. O estabelecimento de *baselines* em sistemas de controle de versões permite reproduzir uma versão anterior do sistema quando isso for desejado. Poder voltar com segurança para uma versão anterior fornece a garantia de um progresso controlado ao longo das atividades de Engenharia de Software.

Dois outros conceitos importantes no contexto da gerência de configuração são o ***build*** e a ***release***. Um *build* representa uma versão não necessariamente completa do sistema em desenvolvimento, mas com certa estabilidade e que pode ser

implantada. Um *build* inclui não só código compilado, mas também a documentação, arquivos de configuração, base de dados etc. Já a *release* se refere a um *build* entregue ao cliente (interno ou externo). No contexto dos métodos ágeis busca-se realizar uma *release* a cada *sprint*. Assim, uma *release* implica no estabelecimento de uma nova *baseline*, permitindo retornar à exata versão dos códigos-fontes daquilo que foi entregue ao cliente em momentos futuros.

Uma boa gerência de configuração requer a integração de ferramentas de gestão de tarefas e de solicitações de mudanças (como o *Jira* ou o *Azure DevOps*) com ferramentas de controle de versões (como o *Git* ou o *SVN*). Isso permitirá saber exatamente quais solicitações de mudança foram atendidas entre uma versão e outra do produto (*Release Notes*) e quais foram as exatas alterações feitas nos arquivos para implementar cada uma das solicitações de mudança.

A gerência de configuração está fortemente ligada com o conceito de DevOps. A seguir, vamos detalhar como esse conceito se manifesta no contexto de sistemas inteligentes.

16.2 DevOps e MLOps no contexto de sistemas inteligentes

DevOps envolve alinhar o desenvolvimento do software com a operação, permitindo aumentar a capacidade de resposta a mudanças por meio de entregas rápidas e de alta qualidade para a operação. É comum que para isso se estabeleça um *pipeline* de integração contínua (CI – *Continuous Integration*) e implantação contínua (CD – *Continuous Deployment*), o que permitirá rapidamente ter as mudanças sendo desenvolvidas integradas no ambiente de desenvolvimento e migrar versões aprovadas com

segurança entre os ambientes de desenvolvimento, teste, homologação e produção.

A preparação deste *pipeline* é normalmente responsabilidade de uma pessoa analista de infra/DevOps e comumente utiliza ferramentas como *AWS CodePipeline*, *Azure Pipelines* e *GitLab CI/CD*, que buscam automatizar o processo de geração de *builds* e a implantação (*deploy*). A seguir, abordaremos três práticas que se mostram importantes no contexto de DevOps e que engenheiros(as) de software e *Machine Learning* deveriam conhecer: **controle de versões**, **integração contínua** e **implantação contínua**. Elas serão descritas nas seções seguintes deste capítulo.

No contexto de sistemas inteligentes, essas práticas se estendem também ao versionamento dos artefatos de *Machine Learning* e à integração e implantação contínuas de alterações feitas nos modelos. Na verdade, para sistemas inteligentes, pode ser necessário ter também um *pipeline* de **MLOps** para retreinar e implantar automaticamente um modelo. Esse *pipeline* adiciona complexidade porque precisa automatizar etapas que cientistas de dados normalmente executam manualmente antes da implantação para treinar e validar novos modelos. As fases típicas de um *pipeline* de MLOps contemplam: coleta e análise de dados, preparação dos dados, treinamento do modelo, validação do modelo, colocar o modelo em operação, monitorar o modelo e retreiná-lo sempre que necessário.

Nem sempre é simples automatizar essas fases por completo, e há situações em que a estratégia de MLOps envolve passos manuais. De fato, MLOps é entendido por alguns como a disciplina que contempla as melhores práticas para colaboração entre cientistas de dados, pessoas desenvolvedoras e profissionais de operação, visando a busca dessa automatização dentro das possibilidades. A automatização é uma meta a ser perseguida e que muitas vezes é alcançável. Diversas das soluções que entregamos recentemente nos projetos da iniciativa ExACTa PUC-Rio, incluindo algumas com pedido

de patente já depositado, contemplam *pipelines* de MLOps completamente automatizados.

A figura a seguir ilustra como se dá em MLOps o alinhamento entre a construção de modelos de *Machine Learning*, o desenvolvimento de software e a operação. O que não transparece na figura é que o modelo pode eventualmente ser reimplantado sem nenhuma alteração no desenvolvimento do sistema que o consome – em particular, no caso em que um modelo é consumido como um *endpoint* na nuvem. Ou seja, podem haver situações em que se aciona somente o *pipeline* de MLOps, mesmo sem alterações no código do restante do sistema e sem a necessidade de acionar o *pipeline* de CI/CD.

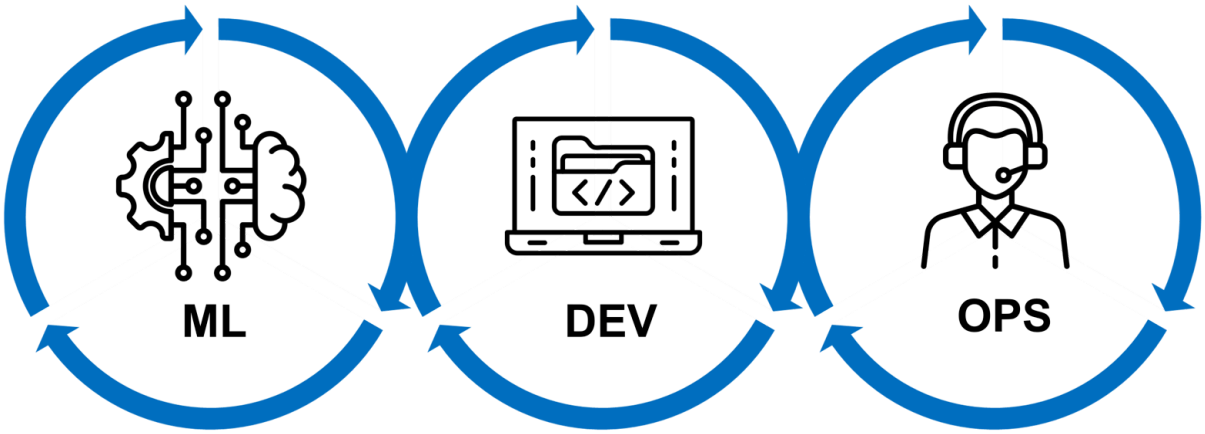


Figura 16.1: MLOps.

Na prática, enquanto o *pipeline* de CI/CD deve ser periodicamente acionado visando uma integração e implantação contínuas das alterações sendo feitas no código, normalmente o *pipeline* de MLOps é disparado a partir da necessidade de uma atualização do modelo, como quando um fenômeno conhecido como *concept drift* é identificado durante a monitoração do modelo. Dizemos que ocorreu um *concept drift* quando mudanças no comportamento dos dados são identificadas. Assim, é comum que na etapa de monitoração do modelo existam mecanismos de identificação de *concept drift*, a fim

de disparar o *pipeline* de MLOps para, por exemplo, iniciar o retreinamento do modelo com novos dados.

16.3 Controle de versões de artefatos de Machine Learning

As ferramentas de controle de versões buscam manter todos os arquivos em um repositório central e controlar o acesso a esse repositório, de modo a garantir a consistência dos artefatos. O uso de uma ferramenta de controle de versão é fundamental quando se busca desenvolver um software em equipe. Hoje em dia, a ferramenta mais utilizada para o controle de versões de projetos de software é o **Git**, mas existem outras ferramentas, como o **DVC** (*Data Version Control*), que é mais voltado para projetos de Ciência de Dados. Nesta seção, explicaremos os conceitos básicos que permitirão utilizar essas ferramentas.

Git

Conforme ilustrado na figura a seguir, o Git funciona com dois repositórios, um local e um remoto. Além disso, existe a área intermediária, conhecida como área de *staging* (*staging area*), que armazena os arquivos que se pretende versionar. Os arquivos da sua pasta de trabalho local podem ser adicionados à *staging area* utilizando o comando *git add*. Alterações em arquivos adicionados na *staging area* podem ser passados para o repositório local utilizando o comando *git commit*. Alterações no repositório local podem ser passadas para o repositório remoto utilizando o comando *git push*. Alterações feitas no repositório remoto por outros desenvolvedores podem ser obtidas utilizando o comando *git pull*. É possível fazer cópias do repositório local para uma pasta de trabalho local utilizando o comando *git checkout*. O comando *git merge* permitirá

integrar mudanças do seu repositório local para a sua pasta de trabalho local.

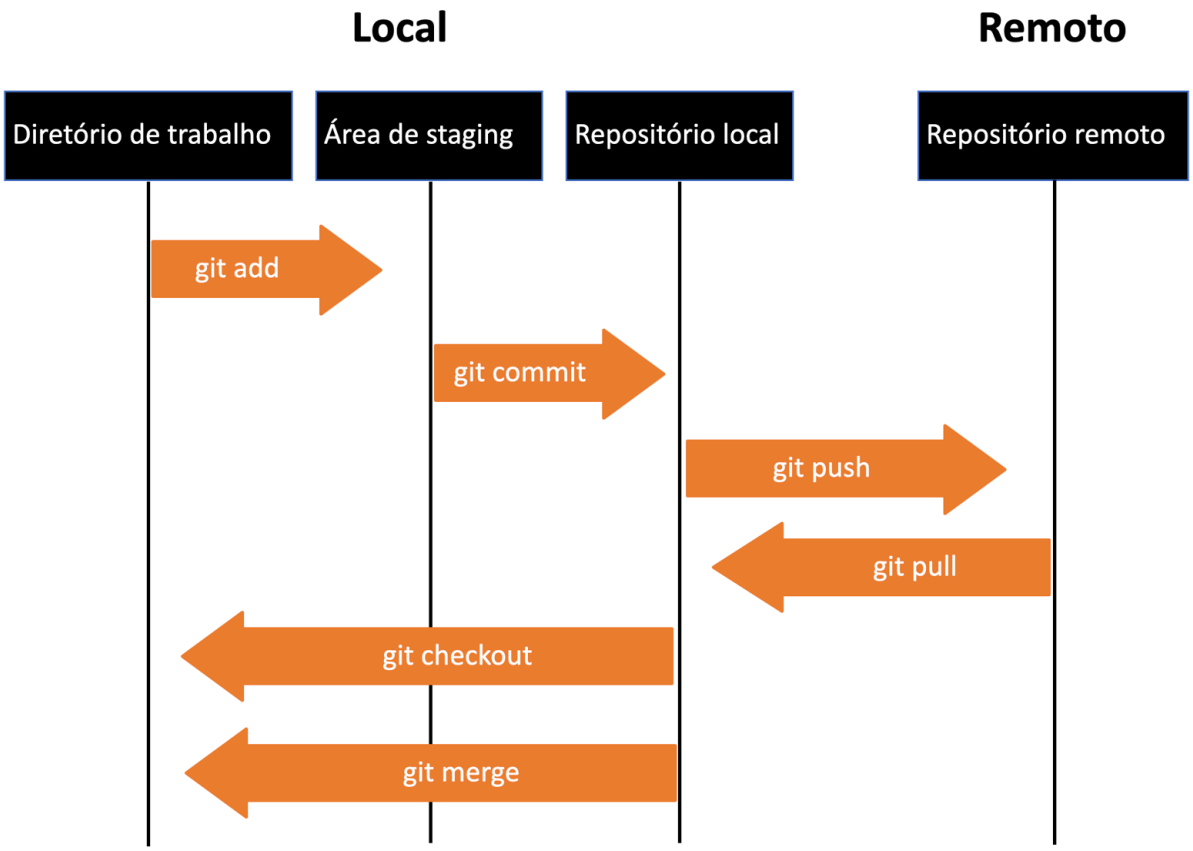


Figura 16.2: Resumo do funcionamento do Git.

Além desses comandos, dois conceitos importantes nesse contexto são o de ***branch*** e ***merge***. Um *branch* representa um fluxo alternativo para atualização de versões de itens de configuração. *Branches* normalmente se originam de correções em versões anteriores. Dessa forma, faz-se um *pull* do repositório remoto daquela *baseline*, um *checkout* do repositório local para então criar um *branch* e realizar as correções de forma que não afete outras versões. Caso a correção se aplique também às versões futuras, será então preciso realizar um *merge* para a integração da mudança no fluxo principal. O *merge* nada mais é do que a unificação de diferentes versões de um mesmo item de configuração, ou seja, a

integração dos itens de configuração de um *branch* com os itens de configuração do fluxo principal (conhecido como *master branch*).

Muitas vezes a integração de um *branch* alternativo em um *master branch* é precedida de um *pull request*, ou seja, uma solicitação de integração. O uso de *pull requests* permite integrar no processo as revisões de código modernas, que detalhamos anteriormente. A figura a seguir ilustra essa integração. Nela, um *branch* alternativo foi criado e, após dois *commits* de alterações nesse *branch*, um *pull request* foi solicitado. As alterações são então revisadas antes que ocorra o *merge*, que integra as alterações (caso aprovadas) no fluxo principal (*master branch*). Para um referencial completo de comandos do Git, sugerimos recorrer à documentação oficial, que se encontra disponível em https://git-scm.com/doc.

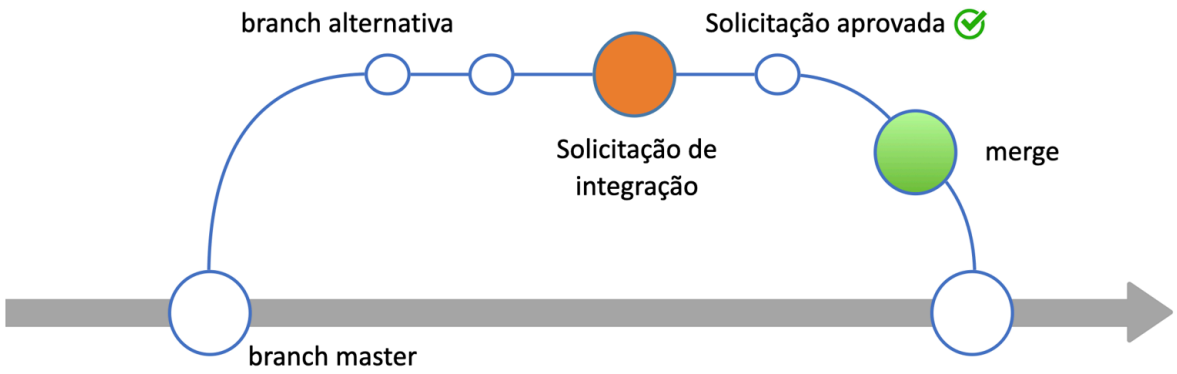


Figura 16.3: Exemplo de integração no Git.

Data Version Control (DVC)

O **DVC** é um sistema de controle de versão de uso livre e código aberto projetado especificamente para gerenciar dados, *pipelines*, modelos de *Machine Learning* e experimentos. Ele permite que cientistas e analistas de dados rastreiem alterações em dados, colaborem com outras equipes e reproduzam resultados. No DVC, os dados são tratados como o principal ativo do projeto, o que significa que são tratados como código em um sistema de controle de versão

Git. Ou seja, os dados, *pipelines*, modelos e experimentos podem ser versionados, rastreados e gerenciados de maneira semelhante ao código.

O DVC funciona armazenando os dados em um repositório central e rastreando as alterações nesses dados ao longo do tempo. Ele usa uma abordagem baseada em arquivo, em que cada arquivo no repositório representa uma parte dos dados ou uma etapa em um *pipeline* de dados. Quando são feitas alterações nos dados ou no *pipeline*, o *DVC* registra essas alterações e as armazena no repositório. Isso permite que os usuários rastreiem o histórico de seus dados, modelos, experimentos e os revertam para versões anteriores, se necessário.

Se você já usou plataformas de *Machine Learning* como o *Microsoft Azure* ou *Google Cloud Platform*, talvez tenha pensado nas funcionalidades de versionamento de *datasets*, modelos e experimentos que elas oferecem. No entanto, essas funcionalidades não estão vinculados diretamente com o código do projeto de Ciência de Dados. Já o *DVC* funciona sobre repositórios Git garantindo o histórico único para dados, código e modelos envolvidos no desenvolvimento de software do projeto de Ciência de Dados. Além disso, o *DVC* é uma ferramenta autônoma que pode ser usada com qualquer plataforma ou fluxo de trabalho de *Machine Learning*. Isso significa que você pode usá-lo para criar versões de dados independentemente da plataforma que estiver usando, oferecendo mais flexibilidade de escolha.

No geral, o *DVC* foi projetado para incorporar as melhores práticas do versionamento de software em fluxos de trabalho de *Machine Learning*. O DVC pode ajudar cientistas e analistas de dados a gerenciar e rastrear seus dados de forma eficiente, permitindo que produzam resultados confiáveis. Para um referencial completo sobre o uso de *DVC*, sugerimos recorrer à documentação oficial, que se encontra disponível em https://dvc.org/doc.

16.4 Integração contínua (CI)

A **integração contínua** consiste na frequente integração das alterações feitas pelos desenvolvedores no repositório remoto e a geração frequente (pelo menos diária) de *builds* do sistema. É comum que esses *builds* sejam submetidos a testes unitários e de integração automatizados, permitindo que problemas de integração possam a ser encontrados logo que introduzidos, na maioria dos casos.

Naturalmente, as predições dos componentes de *Machine Learning* devem ser também testadas como parte desses testes unitários e de integração, seguindo as sugestões do capítulo anterior sobre controle da qualidade. A integração contínua é considerada uma das melhores práticas do desenvolvimento de software. Para uma estratégia eficiente de integração contínua, a geração de *builds* deve ser automatizada e realizada com frequência adequada.

16.5 Implantação contínua (CD)

Enquanto alterações são frequentemente integradas no *branch* principal com a integração contínua, quando se usa **implantação contínua** (*continuous deployment*), as alterações no *branch* principal são rapidamente levadas para a produção. Isso requer um conjunto de testes automatizados para que essa migração automática para a produção possa ser feita com segurança. Para isso, algumas vezes ao dia, o servidor de integração contínua realiza testes mais exaustivos, incluindo testes de sistema, com as alterações que ainda não entraram em produção.

Esses testes naturalmente contemplam testes de sistema de aspectos funcionais e não funcionais das funcionalidades que envolvem *Machine Learning*, de acordo com os requisitos estabelecidos. Se todos os testes passarem, as alterações são então

automaticamente implantadas em produção e disponibilizadas para os usuários.

Um relato de sucesso amplamente conhecido de implantação contínua é o da estratégia DevOps reportada pelo Facebook (atual Meta) (FEITELSON *et al.*, 2013), que permite à empresa, que até hoje é referência no uso de boas práticas de DevOps, implantar novos desenvolvimentos para que os usuários possam utilizar assim que estiverem concluídos. O artigo de Feitelson, Frachtenberg e Beck (2013) reforça as revisões de código modernas e testes automatizados, que discutimos no capítulo anterior, como fatores-chave para o sucesso de estratégias de DevOps.

Como vimos, a gerência de configuração representa um aspecto-chave para o desenvolvimento de software com segurança e produtividade. Uma gerência de configuração eficaz pode ser obtida através do estabelecimento de políticas de gerência de configuração (para o estabelecimento de *baselines*, a criação de *branches* e a geração de *builds* e *releases*, por exemplo) e da integração entre ferramentas, como *issue trackers* e ferramentas de controle de versão. Reforçamos ainda a importância de práticas de DevOps e benefícios do estabelecimento de um *pipeline* de CI/CD e, no contexto de sistemas inteligentes, também de um *pipeline* de MLOps.

Capítulo 17
Referências bibliográficas

ALONSO, Silvio; KALINOWSKI, Marcos; FERREIRA, Bruna; BARBOSA, Simone D. J.; LOPES, Hélio. A systematic mapping study and practitioner insights on the use of software engineering practices to develop MVPs. *Information and Software Technology*, v. 156, 2023. Disponível em: https://www-di.inf.puc-rio.br/~kalinowski/publications/AlonsoKVFB21.pdf. Acesso em: 27 mar. 2023.

ANDERSON, David J. *Kanban*: successful evolutionary change for your technology business. Washington: Blue Hole Press, 2010.

ANICHE, Maurício. *Effective Software Testing*: A Developer's Guide. New York: Manning, 2022.

BACCHELLI, Alberto; BIRD, Christian. Expectations, outcomes, and challenges of modern code review. *In*: *35th International Conference on Software Engineering (ICSE)*, San Francisco, 2013, p. 712-721. 2013. Disponível em: doi: 10.1109/ICSE.2013.6606617. Acesso em: 27 mar. 2023.

BARBOSA, Simone D. J.; SILVA, Bruno D.; SILVEIRA, Milene S.; GASPARINI, Isabela; DARIN, Ticianne; BARBOSA, Gabriel D. J. *Interação humano-computador e experiência do usuário*. Independente, 2021.

BECK, Kent; BEEDLE, Mike; VAN BENNEKUM, Arie; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for agile software development*. (online). Agile Alliance, 2001.

BERRY, Daniel M. Requirements Engineering for Artificial Intelligence: What Is a Requirements Specification for an Artificial Intelligence?. *In: Requirements Engineering*: Foundation for Software Quality (REFSQ). 28th International Working Conference 2022, Birmingham, 2022. Disponível em: https://doi.org/10.1007/978-3-030-98464-9_2. Acesso em: 27 mar. 2023.

BOURQUE, Pierre; FAIRLEY, Richard E. (eds.). SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge. *IEEE Computer Society*, 2014.

Disponível em: https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf. Aesso em 27 mar. 2023.

BUDDEN, Phil; MURRAY, Fiona. *MIT's stakeholder framework for building & accelerating innovation ecosystems*. MIT's Laboratory for Innovation Science & Policy, 2019. Disponível em: https://innovation.mit.edu/assets/MIT-Stakeholder-Framework_Innovation-Ecosystems.pdf. Acesso em: 31 jan. 2023.

BRUCE, Peter; BRUCE, Andrew. *Estatística prática para cientistas de dados*. Rio de Janeiro: Alta Books, 2019.

BROOKS, Frederick P. No Silver Bullet Essence and Accidents of Software Engineering. *In: IEEE Computer*, v. 20, n. 4, p. 10-19, 1987. Disponível em: https://ieeexplore.ieee.org/document/1663532. Acesso em: 23 mar. 2023.

CARLETON, Anita; SHULL, Forrest; HARPER, Erin. Architecting the Future of Software Engineering. *In*: *IEEE Computer*, v. 55, n. 9, p. 89-93, 2022. Disponível em: https://doi.org/10.1109/MC.2022.3187912. Acesso

em: 27 mar. 2023.

CAROLI, Paulo. *Lean Inception*: how to align people and build the right product. Rio de Janeiro: Editora Caroli, 2018.

CERVANTES, Humberto; KAZMAN, Rick. *Designing Software Architectures*: A Practical Approach. 1. ed. Boston: Addison-Wesley, 2016.

COHN, Mike. *Agile estimating and planning*. London: Pearson Education, 2006.

CUNNINGHAM, Ward. The WyCash portfolio management system. *In: ACM SIGPLAN OOPS Messenger*, v. 4, n. 2, p. 29–30, 1992. Disponível em: https://doi.org/10.1145/157710.157715. Acesso em: 27 mar. 2023.

DELAMARO, Márcio E.; MALDONADO, José C.; JINO, Mario. *Introdução ao Teste de Software*. 2. ed. Rio de Janeiro: Elsevier-Campus, 2016.

DEMARCO, Tom; LISTER, Timothy. *Peopleware*: productive projects and teams. 3. ed. Boston: Addison-Wesley, 2013.

DORARD, Louis. *The Machine Learning Canvas*. 2015. Disponível em: https://www.machinelearningcanvas.com/. Acesso em: 27 mar. 2023.

ESCOVEDO, Tatiana; KOSHIYAMA, Adriano. *Introdução a Data Science*: Algoritmos de Machine Learning e métodos de análise. São Paulo: Casa do Código, 2022.

FARLEY, David. *Modern Software Engineering*: Doing What Works to Build Better Software Faster. Boston: Addison-Wesley Professional, 2021.

FEITELSON, Dror G.; FRACHTENBERG, Eitan; BECK, Kent L. Development and Deployment at Facebook. *In: IEEE Internet Computing*, v. 17, n. 4, p. 8-17, 2013. Disponível em: https://ieeexplore.ieee.org/document/6449236. Acesso em: 27 mar. 2023.

FERNANDES, Eduardo; KALINOWSKI, Marcos. On the perceived relevance of critical internal quality attributes when evolving software features. *In: International Conference on Cooperative and Human Aspects of Software Engineering* (CHASE 2023), Melbourne, 2023.

FERNÁNDEZ-SÁEZ, Ana M.; CHAUDRON, Michel R.; GENERO, Marcela. An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles. *In: Empirical Software Engineering*, v. 23, n. 6, p. 3281-3345, 2018. Disponível em: https://link.springer.com/article/10.1007/s10664-018-9599-4. Acesso em: 27 mar. 2023.

FERREIRA, Bruna; MARQUES, Silvio; KALINOWSKI, Marcos; LOPES, Hélio; BARBOSA, Simone D. J. Lessons learned to improve the UX practices in agile projects involving data science and process automation. *In: Information and Software Technology*, v. 155, 2023. Disponível em: https://doi.org/10.1016/j.infsof.2022.107106. Acesso em: 27 mar. 2023.

FITZGERALD, Brian; STOL, Klaas-Jan. Continuous software engineering: A roadmap and agenda. *In: Journal of Systems and Software*, v. 123, p. 176-189, 2017. Disponível em: https://doi.org/10.1016/j.jss.2015.06.063. Acesso em: 27 mar. 2023.

FOIDL, Harald; FELDERER, Michael; RAMLER,

Rudolph. Data smells: categories, causes and consequences, and detection of suspicious data in AI-based systems. *In: 1st International Conference on AI Engineering – Software Engineering for AI* (CAIN), Pittsburgh, 2022. p. 229-239. Disponível em: https://ieeexplore.ieee.org/document/9796422. Acesso em: 27 mar. 2023.

FOWLER, Martin. Inversion of control containers and the dependency injection pattern. *martinfowler.com*, 2004. Disponível em: http://martinfowler.com/articles/injection.html. Acesso em: 27 mar. 2023.

FOWLER, Martin. *UML Essencial*: um breve guia para linguagem padrão. Porto Alegre: Bookman, 2014.

FOWLER, Martin. *Refactoring*. Boston: Addison-Wesley Professional, 2018.

GAMMA, Erich; JOHNSON, Ralph; HELM, Richard; VLISSIDES, John. *Design patterns*: elements of reusable object-oriented software. Boston: Addison-Wesley, 1995.

GÉRON, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. 3. ed.

Sebastopol: O'Reilly Media, 2022.

GIRAY, Görkem. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *In: Journal of Systems and Software*, v. 180, 2021. Disponível em: https://doi.org/10.1016/j.jss.2021.111031. Acesso em: 27 mar. 2023.

GORTON, Ian. *Essential Software Architecture*. 2. ed. Springer, 2011.

GLINZ, Martin; VAN LOENHOUD, Hans; STAAL, Stefan; BÜHNE, Stan. Handbook for the CPRE Foundation Level according to the IREB Standard. *In: International Requirements Engineering Board* (IREB), 2020. Disponível em: https://www.ireb.org/content/downloads/3-cpre-foundation-level-handbook/cpre_foundationlevel_handbook_en_v1.1.pdf. Acesso em: 27 mar. 2023.

ISACA. *CMMI Development*. c2023. Disponível em: https://cmmiinstitute.com/cmmi/dev. Acesso em: 31 dez. 2022.

JACOBSON, Ivar; LAWSON, Harold; NG, Pan-Wei;

MCMAHON, Paul E.; GOEDICKE, Michael. *The essentials of modern software engineering*: free the practices from the method prisons! San Rafael: Morgan & Claypool, 2019.

JAIN, Raj. *The art of computer systems performance analysis*: techniques for experimental design, measurement, simulation, and modeling. Hoboken, New Jersey: John Wiley & Sons, 1990.

JÚNIOR, Ed; FARIAS, Kleinner; SILVA, Bruno. A Survey on the Use of UML in the Brazilian Industry. *In:* Simpósio Brasileiro de Engenharia de Software (SBES), 35., Joinville, 2021. *Anais* (...), p. 275-284. Porto Alegre: Sociedade Brasileira de Computação, 2021. Disponível em: https://doi.org/10.1145/3474624.3474632. Acesso em: 27 mar. 2023.

KALINOWSKI, Marcos; LOPES, Hélio; TEIXEIRA, Alex F.; CARDOSO, Gabriel S.; KURAMOTO, André; ITAGYBA, Bruno; BATISTA, Solon T.; PEREIRA, Juliana A.; SILVA, Thuener; WARRAK, Jorge A.; COSTA, Marcelo; FISCHER, Marinho; SALGADO, Cristiane; TEIXEIRA, Bianca; CHUEKE, Jacques; FERREIRA, Bruna; LIMA, Rodrigo; VILLAMIZAR, Hugo; BRANDÃO, André; BARBOSA, Simone; POGGI,

Marcus; PELIZARO, Carlos; LEMES, Deborah; WALTEMBERG, Marcus; LOPES, Odnei; GOULART, Willer. Lean R&D: An agile research and development approach for digital transformation. *In:* International Conference on Product-Focused Software Process Improvement (PROFES) Turin, 2020. *Proceedings* (...), p. 106-124, 2020. Disponível em: https://link.springer.com/chapter/10.1007/978-3-030-64148-1_7. Acesso em: 27 mar. 2023.

KALINOWSKI, Marcos; CARD, David N.; TRAVASSOS, Guilherme H. Evidence-based guidelines to defect causal analysis. *In: IEEE Software*, v. 29, n. 4, p. 16-18, 2012. Disponível em: https://doi.org/10.1109/MS.2012.72. Acesso em: 27 mar. 2023.

KALINOWSKI, Marcos; WEBER, Kival; SANTOS, Gleison; FRANCO, Nelson; DUARTE, Virginia; TRAVASSOS, Guilherme. Software process improvement results in Brazil based on the MPS-SW model. *In: Software Quality Professional Journal*, v. 17, n. 4, p. 15-28, 2015. Disponível em: https://www-di.inf.puc-rio.br/~kalinowski/publications/KalinowskiWSFDT15.pdf. Acesso em: 27 mar. 2023.

KALINOWSKI, Marcos; MENDEZ, Daniel; GIRAY, Görkem; NEVES, K.; SANTOS-ALVES, A. P.; ESCOVEDO, Tatiana; VILLAMIZAR, Hugo; LAVESON, N.; BALDASSARE, Maria T.; FELDERER, Michael; MUSIL, Juergen; BIFFL, Stefan; LOPES, Hélio; GORSCHEK, Tony. (Submetido). *Naming the Pain in Machine Learning-Enabled Systems Engineering*. 2023.

KUHRMANN, Marco; TELL, Paolo; HEBIG, Regina; KLÜNDER, Jil; MÜNCH, Jürgen; LINSSEN, Oliver; PFAHL, Dietmar; FELDERER, Michael; PRAUSE, Christian; MACDONELL, Stephen G.; NAKATUMBA-NABENDE, Joyce; RAFFO, David; BEECHAM, Sarah; TÜZÜN, Eray; LÓPEZ, Gustavo; PAEZ, Nicolas; FONTDEVILA, Diego; LICORISH, Sherlock; KÜPPER, Steffen; RUHE, Günther; KNAUSS, Eric; ÖZCAN-TOP, Özden; CLARKE, Paul; MCCAFFERY, Fergal; GENERO, Marcela; VIZCAINO, Aurora; PIATTINI, Mario; KALINOWSKI, Marcos; CONTE, Tatiana; PRICKLADNICKI, Rafael; KRUSCHE, Stephan; COSKUNCAY, Ahmet; SCOTT, Ezequiel; CALEFATO, Fabio; PIMONOVA, Svetlana; PFEIFFER, Rolf-Helge; SCHULTZ, Ulrik; HELDAL, Rogardt; FAZAL-BAQAIE, Masud; ANSLOW, Craig; NAYEBI, Maleknaz; SCHNEIDER, Kurt; SAUER, Stefan; WINKLER, Dietmar; BIFFL, Stefan; BASTARRICA, Maria C.;

RICHARDSON, Ita. What Makes Agile Software Development Agile. *In: IEEE Transactions on Software Engineering*, v. 48, n. 9, p. 3523-3539, 2022. Disponível em: https://repositorio.pucrs.br/dspace/bitstream/10923/20122/2/What_Makes_Agile_Software_Development_Agile.pdf. Acesso em: 27 mar. 2023.

LAGE, Luiz C. F.; KALINOWSKI, Marcos; TREVISAN, Daniela; SPÍNOLA, Rodrigo. Usability technical debt in software projects: A multi-case study. *In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM), Porto de Galinhas, Brasil, 2019, p. 1-6. Disponível em: https://www-di.inf.puc-rio.br/~kalinowski/publications/LageKTS19.pdf. Acesso em: 27 mar. 2023.

LAIGNER, Rodrigo; MENDONÇA, Diogo; GARCIA, Alessandro; KALINOWSKI, Marcos. Cataloging dependency injection anti-patterns in software systems. *In: Journal of Systems and Software*, v. 184, 2022. Disponível em: https://doi.org/10.1016/j.jss.2021.111125. Acesso em: 27 mar. 2023.

LAKSHMANAN, Valliappa; ROBINSON, Sara; MUNN,

Michael. *Machine Learning Design Patterns*: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps. Sebastopol: O'Reilly Media, 2020.

LARMAN, Craig. Utilizando UML e padrões: Uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2006.

LEWIS, Grace A.; OZKAYA, Ipek; XU, Xiwei. Software architecture challenges for ML systems. *In: IEEE International Conference on Software Maintenance and Evolution* (ICSME), Luxemburgo, 2021, p. 634-638. Disponível em: https://doi.org/10.1109/ICSME52107.2021.00071. Acesso em: 27 mar. 2023.

MARTIN, Robert C. *Clean Architecture*: A Craftsman's Guide to Software Structure and Design. 1. ed. London: Pearson, 2017.

MENDES, Thiago S.; FARIAS, Mário A. F.; MENDONÇA, Manoel; SOARES, Henrique F.; KALINOWSKI, Marcos; SPÍNOLA, Rodrigo O. Impacts of Agile Requirements Documentation Debt on Software Projects: a Retrospective Study. *In:*

Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Itália, 2016, p. 1290-1295, 2016. Disponível em: http://doi.org/10.1145/2851613.2851761. Acesso em: 27 mar. 2023.

FERNÁNDEZ, Daniel M.; WAGNER, Stefan; KALINOWSKI, Marcos; FELDERER, Michael; MAFRA, Priscilla; VETRO, Antonio; CONTE, Tayana; CHRISTIANSSON, Marie-Therese; GREER, Des; LASSENIUS, Casper; MÄNNISTÖ, Tomi; NAYABI, M.; OIVO, Markku; PENZENSTADLER, Birgit; PFAHL, Dietmar; PRIKLADNICKI, Rafael; RUHE, Günther; SCHEKELMANN, André; SEN, Sagar; SPÍNOLA, Rodrigo O.; TUZCU, Ahmet; DE LA VARA, Jose L.; WIERINGA, Roel. Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical Software Engineering*, v. 22, p. 2298-2338, 2017. Disponível em: https://doi.org/10.1007/s10664-016-9451-7. Acesso em: 27 mar. 2023.

MENDOZA, Isela; KALINOWSKI, Marcos; SOUZA, Uéverton; FELDERER, Michael. Relating Verification and Validation Methods to Software Product Quality Characteristics: Results of an Expert Survey. *In:* International Conference SWQD 2019, 11., Viena,

Áustria, 2019. *Proceedings* (...), 2019. Disponível em: https://doi.org/10.1007/978-3-030-05767-1_3. Acesso em: 27 mar. 2023.

MONDEN, Yasuhiro. Toyota Production System: An Integrated Approach to Just-in-Time. 1. ed. Florida: CRC Press, 1983.

MUNIZ, Antonio; GUILHON, André; GOMES, Cláudio; GASPAR, Eduardo; GUAMÁ, Juliana; CORDEIRO, Karine; ISENSEE, Rodrigo; ESCOVEDO, Tatiana. *Jornada Python*: Uma jornada imersiva na aplicabilidade de uma das mais poderosas linguagens de programação do mundo. Rio de Janeiro: Editora Brasport, 2022.

NEAL ANALYTICS. *Machine Learning Operations (MLOps)*. c2022. Disponível em: https://nealanalytics.com/expertise/mlops/. Acesso em: 31 dez. 2022.

OHNO, Taiichi. *Toyota Production System*: Beyond Large-Scale Production. Florida: CRC Press, 1988.

OZKAYA, Ipek. What Is Really Different in Engineering AI-Enabled Systems? *In*: *IEEE Software*, v. 37, n. 4, p. 3-6, 2020. Disponível em:

https://doi.org/10.1109/MS.2020.2993662. Acesso em: 27 mar. 2023.

PETRE, Marian. UML in practice. *In:* ICSE '13 - International Conference on Software Engineering, 35., San Francisco, 2013. *Proceedings* (...), p. 722-731, 2013. Disponível em: https://doi.org/10.1109/icse.2013.6606618. Acesso em: 27 mar. 2023.

PIMENTEL, João F.; MURTA, Leonardo; BRAGANHOLO, Vanessa; FREIRE, Juliana. Understanding and improving the quality and reproducibility of Jupyter notebooks. *Empirical Software Engineering*, v. 26, n. 4, art. 65, 2021. Disponível em: https://doi.org/10.1007/s10664-021-09961-9. Acesso em: 27 mar. 2023.

POPPENDIECK, Mary; POPPENDIECK, Tom. *Lean software development*: an agile toolkit. Boston: Addison-Wesley, 2003.

PRIKLADNICKI, Rafael; WILLI, Renato; MILANI, Fabiano. *Métodos ágeis para desenvolvimento de software*. Porto Alegre: Bookman, 2014.

QUARANTA, Luigi; CALEFATO, Fabio; LANUBILE,

Filippo. Pynblint: a static analyzer for Python Jupyter notebooks. *In:* 2022 IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN), Pittsburgh, 2022. *Proceedings* (...), p. 48-49, 2022. Disponível em: https://collab.di.uniba.it/fabio/wp-content/uploads/sites/5/2022/10/3522664.3528612.pdf. Acesso em: 27. mar. 2023.

RIES, Eric. *The Lean Startup*. New York: Crown Business, 2011.

RIOS, Nicolli; NETO, Manoel; SPÍNOLA, Rodrigo. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, v. 102, p. 117-145, 2018. Disponível em: https://doi.org/10.1016/j.infsof.2018.05.010. Acesso em: 27 mar. 2023.

RODRIGUES, Ariane; BARBOSA, Gabriel; LOPES, Hélio; BARBOSA, Simone. What questions reveal about novices' attempts to make sense of data visualizations: Patterns and misconceptions. *Computers & Graphics*, v. 94, p. 32-42, 2021. Disponível em: https://doi.org/10.1016/j.cag.2020.09.015. Acesso

em: 27 mar. 2023.

RUBIN, Kenneth. *Scrum Essencial*: Um guia prático para o mais popular processo ágil. Rio de Janeiro: Alta Books, 2018.

SADOWSKI, Caitlin; SÖDERBERG, Emma; CHURCH, Luke; SIPKO, Michal; BACCHELLI, Alberto. Modern code review: a case study at google. *In: Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track*, p. 181-190, 2018. Disponível em: https://sback.it/publications/icse2018seip.pdf. Acesso em: 27 mar. 2023.

SADOWSKI, Caitlin; ZIMMERMANN, Thomas. *Rethinking productivity in software engineering*. New York: Apress (Springer), 2019.

SAFe® 5 for Lean Enterprise. (s. d.). Disponível em: https://v5.scaledagileframework.com/. Acesso em: 31 dez. 2022.

SCULLEY, D.; HOLT, Gary; GOLOVIN, Daniel; DAVYDOV, Eugene; PHILLIPS, Todd; EBNER, Dietmar; CHAUDHARY, Vinay; YOUNG, Michael; CRESPO, Jean-François; DENNISON, Dan. Hidden

technical debt in machine learning systems. *In: Proceedings of the 28th International Conference on Neural Information Processing Systems*, v. 2, p. 2503–2511, 2015. Disponível em: https://dl.acm.org/doi/10.5555/2969442.2969519. Acesso em: 27 mar. 2023.

SCULLEY, D.; HOLT, Gary; GOLOVIN, Daniel; DAVYDOV, Eugene; PHILLIPS, Todd; EBNER, Dietmar; CHAUDHARY, Vinay; YOUNG, Michael. Machine Learning: The High-Interest Credit Card of Technical Debt. *In: Software Engineering for Machine Learning* - SE4ML (NeurIPS 2014 Workshop), 2014. Disponível em: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43146.pdf. Acesso em: 27 mar. 2023.

SOFTEX. *MPS.BR - Melhoria de Processo do Software Brasileiro*. Guia Geral MPS de Software – Versão janeiro/2021. 2021. Disponível em: https://softex.br/download/guia-geral-de-software-2021/. Acesso em: 27 mar. 2023.

SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson Universidades, 2019.

SOMMERVILLE, Ian. *Engineering Software Products*:

An Introduction to Modern Software Engineering. 1. ed. London: Pearson, 2019.

SUN, Youcheng; WU, Min; RUAN, Wenjie; HUANG, Xiaowei; KWIATKOWSKA, Marta; KROENING, Daniel. Concolic testing for deep neural networks. *In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, p. 109-119, 2018. Disponível em: https://doi.org/10.1145/3238147.3238172. Acesso em: 27 mar. 2023.

TANG, Yiming; KHATCHADOURIAN, Raffi; BAGHERZADEH, Mehdi; SINGH, Rhia; STEWART, Ajani; RAJA, Anita. An empirical study of refactorings and technical debt in Machine Learning systems. In ICSE '21: Proceedings of the 43rd International Conference on Software Engineering, p. 238-250, 2021. Disponível em: https://doi.org/10.1109/ICSE43902.2021.00033. Acesso em: 27 mar. 2023.

TOM, Edith; AURUM, Aybüke; VIDGEN, Richard. An exploration of technical debt. *In: Journal of Systems and Software*, v. 86, n. 6, p. 1498-1516, 2013. Disponível em: https://doi.org/10.1016/j.jss.2012.12.052. Acesso

em: 23 mar. 2023.

TRAVASSOS, Guilherme; KALINOWSKI, Marcos. *iMPS 2013*: Evidências sobre o desempenho das empresas que adotaram o modelo MPS-SW. São Paulo: Softex, 2014.

VALENTE, Marco T. *Engenharia de Software Moderna*: Princípios e práticas para desenvolvimento de software com produtividade. Independente, 2020.

VAN OORT, Bart; CRUZ, Luís; ANICHE, Maurício; VAN DEURSEN, Arie. The prevalence of code smells in machine learning projects. *In: 2021 IEEE/ACM 1st Workshop on AI Engineering* - Software Engineering for AI (WAIN), Madri, 2021, p. 1-8. Disponível em: https://arxiv.org/pdf/2103.04146. Acesso em: 27 mar. 2023.

VILLAMIZAR, Hugo; KALINOWSKI, Marcos; LOPES, Hélio. A Catalogue of Concerns for Specifying Machine Learning-Enabled Systems. *In*: *Workshop on Requirements Engineering* (WER 2022), Natal, Brasil, 2022a. Disponível em: http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER22/WER_2022_Camera_ready_paper_33.pdf. Acesso em: 27 mar. 2023.

VILLAMIZAR, Hugo; KALINOWSKI, Marcos; LOPES, Hélio. Towards Perspective-based Specification of Machine Learning-Enabled Systems. *In: 48th Euromicro Conference Series on Software Engineering and Advanced Applications* (SEAA), Gran Canaria, Espanha, 2022b. Disponível em: https://arxiv.org/pdf/2206.09760.pdf. Acesso em: 27 mar. 2023.

VILLAMIZAR, Hugo; KALINOWSKI, Marcos; LOPES, Hélio. (Submetido). PerSpecML: A Perspective-based Approach for Specifying Machine Learning-Enabled Systems. *In*: *International Conference on Requirements Engineering*, 2023.

VILLAMIZAR, Hugo; ESCOVEDO, Tatiana; KALINOWSKI, Marcos. Requirements Engineering for Machine Learning: A Systematic Mapping Study. *In: 47th Euromicro Conference on Software Engineering and Advanced Applications* (SEAA), Palermo, Itália, 2021, p. 29-36. Disponível em: https://www-di.inf.puc-rio.br/~kalinowski/publications/VillamizarEK21.pdf. Acesso em: 27 mar. 2023.

WAGNER, Stefan. *Software Product Quality Control*. Berlin: Springer, 2013.